UNIVERSITY OF
CAMBRIDGE

Privacy-preserving decentralised collaborative applications

Stephan Alexander Kollmann

Robinson College

December 2018

This dissertation is submitted for the degree of Doctor of Philosophy.

Abstract

Cloud-based applications are problematic from a privacy perspective because they typically have access to large amounts of user data and metadata. This centralisation of user data creates an attractive target for actors such as criminals, suppressive governments, and companies selling the data. At the same time, the popularity of mobile and web applications has led to a growing amount of sensitive data being stored in the cloud.

This dissertation focuses on collaborative applications, such as Google Docs and Microsoft Office Online, where users currently rely on cloud-based solutions. It explores decentralised alternatives that allow the use of end-to-end encryption and anonymous communication systems to improve both information privacy and communication privacy.

One approach for a collaborative application to synchronise data in a privacy-preserving way is to use Tor hidden services, providing end-to-end encrypted communication, while also hiding collaborators' identity. However, running Tor comes at a cost. We explore the costs of running a hidden service on a smartphone. Smartphones are nowadays the most frequently used computing devices, but they are also relatively resource-constrained. We build an empirical model of monthly cellular data traffic, and estimate a median 198 MiB for a typical user. We further estimate that the network activity would cost at least 9.6% of daily battery capacity on a Nexus One using 3G Internet. We explore four optimisations that, in combination, reduce the estimated median data cost to 61 MiB.

We also consider the security and privacy properties of decentralised collaborative applications, and explore a challenge that is introduced by a decentralised design – the lack of a trusted server guaranteeing consistency between collaborators. We present a novel snapshot protocol that ensures consistency, whilst allowing the past edit history to be hidden from new collaborators, and without relying on a consensus mechanism.

Lastly, we evaluate the overhead of the snapshot protocol by replaying editing histories from 270 Wikipedia articles, and demonstrate how its correctness and security properties are achieved. Assuming the number of collaborators remains small, the protocol is scalable in terms of CPU, memory, and network usage. It substantially reduces the amount of data transferred to a new collaborator compared to a basic protocol that transmits the full history. The computational cost is in the order of milliseconds per operation, indicating the protocol is suitable for applications where the rate of edits is relatively low.

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. It is not substantially the same as any that I have submitted, or am concurrently submitting, for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my dissertation has already been submitted, or is being concurrently submitted, for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. This dissertation does not exceed the prescribed limit of 60 000 words.

Stephan Alexander Kollmann
December, 2018

Acknowledgements

Many people and institutions supported me throughout my PhD. I would like to acknowledge the most significant ones here.

First, I thank Microsoft Research, The Boeing Company, and the Computer Laboratory for generously providing funding for my PhD. I also give sincere thanks to Alastair Beresford, who has been an excellent supervisor, and who always took time to provide guidance and advice when needed. I further thank Andy Rice, Robert Watson, and Jon Crowcroft, whose advice helped me choose the right problems to tackle.

I would also like to express my gratitude to Martin Kleppmann for many insightful discussions and feedback relating to the work on authenticated snapshots. I thank Alfredo Mazzinghi, Diana Vasile, Jiexin Zhang, and the Digital Technology Group for many interesting conversations and for enriching my time at the Computer Laboratory. I thank Jeunese Payne and Christian O'Connell for proofreading. Lastly, I am grateful for the continuous support of my family, and of my friends, particularly Jeunese Payne, Leonhard Helminger, Stefan Czimek, and Vahan Hovhannisyan.

Contents

CHAPTER 1

Introduction

Users have been moving more and more from stationary computers to mobile devices like laptops, smartphones, tables, and smartwatches. In the UK, smartphones are now the most popular device for accessing the Internet [Int18]. Users often own multiple devices, and synchronize their data between devices using the cloud. Cloud services, and the increasing computational power and sensor hardware on devices, have increased convenience and opened up many possibilities. The amount of sensitive data generated, stored, and processed there increases the importance of effective security measures to protect the data on the devices, in the cloud, and in transport.

Significant effort has been put into building better tools and promoting best practices for security and data protection. Techniques such as sandboxing, permission models, code signing, compartmentalization, and trusted execution environments attempt to ensure the integrity of executed code, limit the permissions of untrusted code, mitigate the effects of malicious code or an exploited vulnerability, or protect sensitive data. Developers and applications rely increasingly on Transport Layer Security (TLS) for encrypted data exchange, supported by mechanisms such as Certificate Transparency, HTTP Strict Transport Security, and Public Key Pinning. Multi-factor authentication has raised the bar for hijacking sensitive accounts such as online banking or email. Additionally, more effective distribution of security patches, and rejection or blacklisting of malicious applications in app stores, decrease the impact of vulnerabilities after they are discovered.

However, an important vulnerability remains. Storing large amounts of sensitive data in one place makes makes the service provider an attractive target for internal and external attackers, and a data breach can have disastrous consequences. No computer system is perfectly secure, and numerous high-profile breaches show that even large companies are susceptible to attacks. To name a few examples: in 2013, attackers stole personal information on all 3 billion user accounts at Yahoo [Per17]; in 2014, details of 83 million

customers were affected by a breach at JPMorgan Chase [AHF14]; personal data of 57 million Uber customers and drivers was compromised in 2016 [Kho17].

Storing data in the cloud can also be problematic in countries where service providers are obliged to hand over data to law enforcement and intelligence agencies without appropriate judicial oversight. For certain groups of people who deal with sensitive data, such as medical professionals, lawyers, journalists, diplomats, engineers, activists, and others, this trust model is particularly problematic for legal, ethical, business, and personal safety reasons.

The Snowden revelations in 2013 about NSA mass surveillance of innocent citizens and industrial espionage have led to increased public awareness of the limitations of the protection of people's personal data. Since then, many researchers, developers, and companies have worked on building end-to-end encrypted communication solutions. While not a new idea – protocols such as PGP and OTR had been around for many years – existing solutions were often difficult to use correctly and suffered from low adoption rates. Today, messaging applications such as Signal, WhatsApp, and Apple iMessage, have made end-to-end encrypted messaging and file transfers the default for more than a billion users.

While end-to-end encryption has raised the standards for communication privacy, it only solves part of the problem. Because even encrypted data is typically processed by central servers, service providers still collect large amounts of metadata about users, such as IP addresses, locations, contacts, and communication patterns. This kind of data can be very revealing, and can be processed more efficiently at scale compared to message contents, which makes it a prime target for intelligence agencies, oppressive governments, advertisers, and other companies. To tackle this problem, systems for anonymous and censorship-resistant messaging, web browsing, and file sharing – such as Mixminion, Tor, Freenet, or Loopix – have been developed. Similarly, cryptographic techniques have been applied to build decentralised currencies such as Bitcoin, and others that aim for additional anonymity such as Zcash and Monero.

Collaborative applications, such as document editors, shared calendars, or synchronized address books allow one or more users to edit a document or dataset from multiple devices concurrently. Concurrent changes are merged, ideally without loss of data or creating conflicts that need manual intervention. Such systems maintain a local copy of the data structure on every collaborating user's device. Whenever the user makes an edit on a device, these changes are first applied to the local copy, and then an edit message containing the change is sent to a central server. In widely deployed collaborative document editing applications, such as Google Docs, when the server receives document edit messages, it modifies these messages to assist devices in resolving potentially conflicting concurrent edits, and propagates these updated messages to the devices used by all collaborating users.

Therefore, while communication between user devices and the server can be encrypted, the server itself must be able to read and modify messages, and thus must be trusted to maintain confidentiality and integrity of a cleartext copy of the document. Furthermore, all updates to a document must be shared via the central server, and the server can thus also capture details on the changes that devices perform on the document. For these reasons, the current generation of collaborative document editing systems do not support end-to-end encryption between user devices.

What makes it difficult to build collaborative applications with end-to-end encryption? When a document is edited by a single user on one device at a time, storing the document encrypted on an untrusted server is relatively straightforward using traditional file-level encryption. The problem becomes more complicated when multiple users can edit the document concurrently. This requires either locking the file, or parts of the file, when a user is working on it, or using a different approach that allows concurrent changes to be merged in a way that is consistent across devices without manual intervention. The need for manual merging is accepted in some collaboration tools, e.g. version control systems for software development. However, it is usually undesirable for real-time collaborative applications such as document editing because resolving conflicts would be disruptive.

A number of algorithms and data structures have been developed that allow concurrent edits on a document, and provide a way of consistently applying these across devices. These can be grouped into two approaches: *Operational Transformation* (OT), and *Conflict-free Replicated Data Types* (CRDTs). The former is used in most established collaborative editing applications, including Google Docs. OT algorithms tend to be relatively complex, and most of them rely on a central server. CRDTs are a more recent approach; they tend to be simpler, and they support peer-to-peer communication between devices because they do not require a total ordering of operations. This also makes them suitable for end-to-end encryption, by sending messages using a suitable secure messaging protocol.

A peer-to-peer based model is also desirable for limiting the amount of metadata collected by a central server. However, having devices communicate directly brings some practical challenges. First of all, mobile devices usually do not have publicly addressable IP addresses, so building a direct communication channel is not straightforward. Second, devices may frequently be offline or have unreliable network connections. Lastly, without a trusted server, devices need to ensure in a decentralised way that their copies of the document are consistent, even in the presence of malicious or malfunctioning devices. Furthermore, since devices may frequently or permanently be offline, they should be able to make progress without requiring a certain proportion of devices to be online.

This dissertation presents and evaluates potential solutions to these challenges. First, it evaluates the use of Tor hidden services as a decentralised privacy-preserving way for mobile devices to communicate. Second, it presents a novel protocol that allows devices to

ensure they are in a consistent state, even in the presence of malicious or malfunctioning devices, while allowing the past editing history to be hidden from new collaborators that are added to a shared document. Lastly, it evaluates the costs of the presented protocol and shows how the protocol achieves its integrity and privacy properties.

1.1 Dissertation outline

Chapter 2 introduces the necessary background for later chapters. It describes different approaches for collaborative editing; cryptographic tools such as accumulators, redactable signatures, and Merkle trees; and systems for anonymous communication, in particular the Tor anonymity network and its hidden services.

Chapter 3 describes the kind of collaborative applications we envision. It further defines the scope of this dissertation, our assumptions, and the threat model.

Chapter 4 discusses the use of Tor and its hidden services to provide a privacy-preserving means of communication between smartphones that does not rely on a centralised server infrastructure. It evaluates the costs of such a solution for typical usage patterns of smartphones running a hidden service, in particular in terms of cellular data traffic, and provides an estimate for its battery usage.

Chapter 5 introduces the concept of authenticated snapshots for collaborative applications. Snapshots allow new users or devices to join a group of collaborators and to start collaborating from the current state of the shared document or data set without receiving the full history of edits. We present a scalable protocol that provides this feature, while still allowing a new collaborator to verify the consistency of the snapshot with the view of other honest collaborators. This allows existing collaborators to hide past edits from new collaborators and can decrease the communication overhead when joining the group, while not sacrificing consistency.

Chapter 6 evaluates the costs of the protocol introduced in Chapter 5, in particular in terms of the computational and communication overhead. Furthermore, it explains how the protocol fulfils the design goals and security and privacy properties stated in Chapter 3, and provides a proof for the consistency property which requires more elaborate discussion.

Chapter 7 concludes the dissertation by summarising the work and results and providing some directions for future work.

1.2 Publications and contributions

What follows is a list the publications I co-authored during the course of my PhD.

1. Stephan A. Kollmann, and Alastair R. Beresford. "The Cost of Push Notifications for Smartphones using Tor Hidden Services." In *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*.

   This publication forms the basis of Chapter 4 of this dissertation. The original idea emerged from discussions with Alastair. The experiments were designed by myself in collaboration with Alastair. I implemented all the necessary code for the instrumentation of Tor, the experiment setup, and the data analysis. I executed the experiments, and collected and analysed the data for the evaluation. I thank: Alastair for help with writing; Nikilesh Balakrishnan, Lucian Carata, and Ripduman Sohan for their assistance with the experimental method; and Martin Kleppmann, Laurent Simon, Daniel R. Thomas, and Diana Vasile for helpful discussion and insight.

2. Martin Kleppmann, Stephan A. Kollmann, Diana A. Vasile, and Alastair R. Beresford. "From secure messaging to secure collaboration." In *26th International Workshop on Security Protocols* (2018).

   This publication is a position paper. Ideas from it are contained in Chapters 1, 3 and 5. It was the result of many hours of discussion between all authors, and written primarily by Martin.

3. Stephan A. Kollmann, Martin Kleppmann, and Alastair R. Beresford. "Snapdoc: Authenticated snapshots with history privacy in peer-to-peer collaborative editing". To appear in *Proceedings on Privacy Enhancing Technologies 2019 (3)*.

   This publication forms the basis of Chapters 5 and 6. Martin conceived the original idea of authenticated snapshots. I designed the protocol for snapshots, and implemented the prototype and the code for the experiment setup and data analysis. I further executed the experiments and collected and analysed the data for evaluation. I conducted the theoretical analysis of the protocol properties in Chapter 6. Throughout, Alastair and Martin contributed with useful discussions and feedback, and assisted in writing; Martin helped in particular with writing up the formal description of the protocol. I thank Ricardo Mendes, Laurent Simon, Tom Sutcliffe, Daniel R. Thomas, Diana Vasile, and Jiexin Zhang for helpful discussions and insight.

CHAPTER 2

Background

Secure, privacy-preserving decentralised collaborative editing requires a number of building blocks. This chapter explains the background for the components required for later chapters. First, appropriate algorithms and data structures that allow users to edit a document concurrently from multiple devices are needed. Such techniques are discussed in Section 2.1. Since we aim for a decentralised system, and therefore do not want to rely on central servers, at least some of the devices need to be able to exchange messages between themselves directly. This by itself is not straightforward, since most consumer devices, especially mobile devices, are behind firewalls and routers using Network Address Translation (NAT), and are therefore not directly addressable. Additionally, for a privacy-preserving system, it can be desirable to hide which users are collaborating, even from someone observing the network traffic. One way to address both issues is to use Tor hidden services, which give each device a globally addressable unique identifier, and provide end-to-end encryption and anonymity to devices (under a certain threat model). Tor and Tor hidden services are described in Section 2.6.1, and the costs for this approach are discussed in Chapter 4.

Traditional centralised collaborative applications use the central server not only as a central point of communication, but clients also put a substantial amount of trust into the server. They tend to rely on the server to enforce access control policies, to protect confidentiality and integrity of the data, and to ensure that all devices have a consistent view of the shared document. When a new device is added as a collaborator, in a centralised system, the server can provide the latest state of the document to the new device. Usually, the server is assumed to be trusted, so there is no mechanism for devices to verify that their view of the document is consistent. In a decentralized system, devices can use secure messaging protocols to protect confidentiality and integrity of the shared data from non-collaborators. However, devices need to either trust the information they get from other collaborators, or the system needs to provide a way for them to verify that

their view is consistent with other devices', e.g. to prevent a collaborator from claiming that another collaborator has written something they never have. In principle, this can be achieved using cryptographic hashes and digital signatures, or using consensus algorithms. However, things are less straightforward if there are no guarantees about devices being online, and if new devices only receive the latest state of a document, and not the editing history. This problem is considered in more detail in Chapter 5. The solution presented there relies on constructs such as cryptographic accumulators (Section 2.3) and redactable signatures (Section 2.4).

2.1 Collaborative editing

In (real-time) collaborative editing systems, each client typically keeps a *replica* of a shared document or data set that is replicated between clients asynchronously. To be considered usable, collaborative systems need to fulfil a number of additional properties compared to single-user applications. First, a responsive user interface requires local changes to be applied with minimal delays. Second, this should be achieved without using locks on the document or parts of it, since locks require consensus between devices and therefore cause delays, or even unavailability of the service in case of a network partition. Third, replicas need to stay consistent. Since processing, network delays, and network partitions make it impossible for replicas to always stay perfectly in sync, a weaker notion of consistency is required. A commonly used consistency model is the CCI model [SJZ+98]. The CCI model requires three properties (properties and the definition below are taken from the original paper):

- **C**onvergence: When the same set of operations have been executed at all sites, all copies of the shared document are identical.
- **C**ausality preservation: For any pair of operations O_a and O_b, if O_a has *happened before* O_b, then O_a is executed before O_b at all sites.
- **I**ntention preservation: For any operation O, the effects of executing O at all sites are the same as the intention of O, and the effect of executing O does not change the effects of independent operations.

The causality preservation property above relies on the happened-before relation on operations, which is defined as follows.

Definition 2.1. Causal ordering relation (happened-before relation) "$\prec$". Given two operations O_a and O_b, generated at sites i and j, then $O_a \prec O_b$, if and only if (1) $i = j$ and the generation of O_a happened before the generation of O_b; or (2) $i \neq j$, and the execution of O_a at site j happened before the generation of O_b; or (3) there exists an operation O_x,

such that $O_a \prec O_x$ and $O_x \prec O_b$. Two operations are *concurrent* if neither $O_a \prec O_b$, nor $O_b \prec O_a$.

The last property, intention preservation, leaves room for interpretation, and the interpretation depends on the application. For character-wise text editing, the usual interpretation is that each inserted character can be assigned a unique identifier; if a character x is added to a string abc between a and b, it ends up between a and b on all replicas, even if other operations happen concurrently; if a character is deleted on at least one replica, it is deleted from all replicas. Attiya et al. [ABG+16] give a precise specification of a replicated list data type to capture this intention. Essentially, they require that list elements have a globally unique identifier, and a total order exists over all elements that is consistent with the list order.

Two main approaches have been proposed for real-time collaborative editing. The traditional one is Operational Transformation; more recently, Conflict-free Replicated Data Types have been proposed. Sections 2.1.1 and 2.1.2 explain the two approaches. Section 2.1.3 briefly discusses how version control systems fit into this space.

2.1.1 Operational Transformation (OT)

Consider a text document shared between two users, initially containing the string "bce". User 1 inserts 'a' at the beginning of the string (changing the string to "abce"), and concurrently, User 2 inserts 'd' at position 3 (changing the string to "bcde"). Now both users send their edits to each other. When User 2 applies the edit by User 1 and inserts 'a' at position 1, the end up with the string "abcde". However, if User 1 directly applies User 2's edit, she will end up with "abdce", diverging from User 2. Operational Transformation (OT) systems solve this problem by transforming concurrent operations against each other. In the above example, User 1 would transform User 2's operation against her own, changing the insert position from 3 to 4. The example is illustrated in Figure 2.1.

Generally, OT systems contain two key components: the transformation functions that transform functions against each other, and the control algorithms that control the transformation process. To achieve convergence when concurrent operations may be executed in arbitrary orders, two properties need to be fulfilled in OT systems:

Convergence Property 1 (CP1). Given two concurrent operations O_a and O_b on the document state s, the transformation function T preserves CP1 if:

$$s \circ O_a \circ T(O_b, O_a) = s \circ O_b \circ T(O_a, O_b).$$

This means that applying O_a followed by O_b transformed against O_a on a state s produces the same state as applying O_b followed by O_a transformed against O_b.

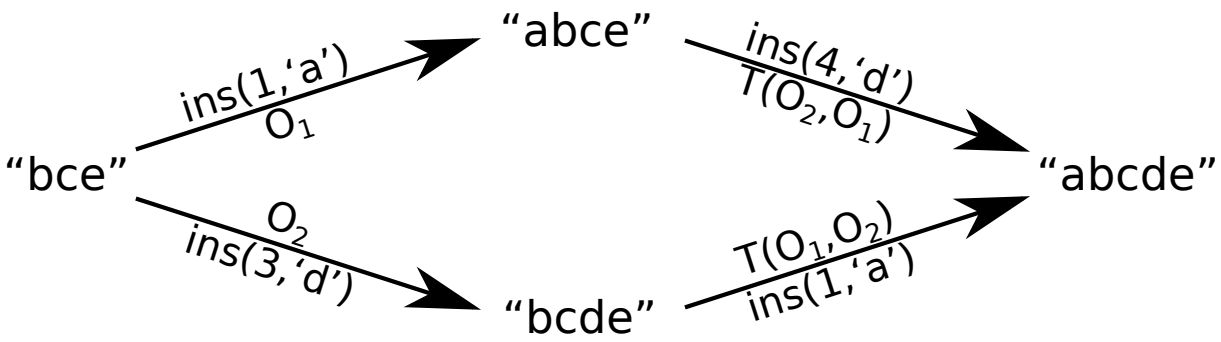


Figure 2.1: Example of concurrent editing of a document initially containing the string "bce", using OT. One user adds 'a' at the beginning, while another user adds 'd' after 'c'. The users then exchange their edit operations, and each user transforms the other user's operation against their own before applying it locally.

Convergence Property 2 (CP2). Given three concurrent operations O_a, O_b and O_c on the document state s, T preserves CP2 if:

$$T(T(O_c, O_a), T(O_b, O_a)) = T(T(O_c, O_b), T(O_a, O_b)),$$

meaning that transforming O_c against O_a and then against $T(O_b, O_a)$ produces the same operations as transforming O_c against O_b and then against $T(O_a, O_b)$.

While it is relatively easy to design transformation functions that satisfy CP1, it turned out to be surprisingly difficult to preserve CP2. A number of proposed OT algorithms were later shown to violate CP2 and therefore cause divergence under certain circumstances [IMOR03, IROM06, OUMI05]. Therefore, instead of preserving CP2, many OT systems opt for control algorithms that constrain when operations are transformed against each other. To achieve this, to the best of our knowledge, all existing OT systems that avoid the need to satisfy CP2 rely on an implicit or explicit total order of operations [XS16]. This is often achieved using a central server that either serializes all operations before forwarding them, as in Jupiter [NCDL95] or Apache Wave (formerly Google Wave) [WMLT15], or using a central sequencer that provides sequence numbers, as in SOCT3 and SOCT4 [VCFS00]. However, there also exist OT systems that use distributed schemes to achieve a total ordering, for example TIBOT [LLS04].

2.1.2 Conflict-free Replicated Data Types (CRDTs)

Conflict-free Replicated Data Types (CRDTs) are an alternative approach for real-time collaborative editing. CRDTs do not need conflict resolution or transformation of operations, because operations are designed such that concurrent operations can be applied conflict-free in any order. Updates at a replica are applied locally immediately without any synchronization, and are broadcast asynchronously to other replicas. All replicas converge

to the same state provided that they receive all updates eventually. More formally, CRDTs provide *strong eventual consistency* [SPBZ11b] while remaining available under arbitrary network partitions (in the sense of the CAP theorem [Bre00, GL02]). The following definition is taken from Shapiro et al. [SPBZ11b].

Definition 2.2. Strong eventual consistency. An object is *strongly eventually consistent* if the following properties hold:

Eventual delivery: An update delivered at some correct replica is eventually delivered to all correct replicas.

Strong convergence: Correct replicas that have received the same updates have equivalent state.

Termination: All method executions terminate.

CRDTs have been proposed for counters, registers, sets, maps, lists [RJKL11, SPBZ11a, SPBZ11b], XML documents [MUW10], and JSON [KB17a]. A shared text document can be stored and replicated using any list CRDT. However, a number of CRDTs have been proposed that are especially optimised for collaborative editing [PMSL09, NMMD13, WUM09, WUM10].

There are two main types of CRDTs: state-based, and operation-based. The following sections give an overview of both.

2.1.2.1 State-based CRDTs

A state-based CRDT provides three methods that can be executed on a replica:

- An *update* method that updates the local state;
- a *query* method that allows reading the local state; and
- a *merge* method that merges a state received from another replica into the local state.

Each replica broadcasts their state to other replicas, e.g. after every update or periodically.

Example: Counter An increment-only replicated counter can be implemented by keeping a vector $(c_1, ..., c_n)$ as the state, where n is the number of replicas. To increment the counter on replica i, increment c_i. Merging states is done by taking the element-wise maximum of the states. To retrieve the current counter value, sum up the vector elements.

This counter does not support decrementing in the same way as incrementing since merging with a lower value has no effect. However, a counter allowing a decrement

operation can be implemented using a second increment-only counter that counts the number of decrements.

Since state-based CRDTs broadcast the entire state, they are usually not well suited for applications like real-time collaborative document editing, where transmitting the entire document for every change would cause a large overhead.

2.1.2.2 Operation-based CRDTs

Operation-based (or op-based) CRDTs do not have a merge method. Instead, after applying an update locally, a replica broadcasts enough information to allow other replicas to replay the update. Assuming delivery of messages in causal order, a sufficient property for an operation-based CRDT is the commutativity of any concurrent operations. An op-based CRDT with this commutativity property is also called Commutative Replicated Data Type (CmRDT). The following examples describe two CmRDTs, Treedoc and RGA; Treedoc is a basis for the protocol presented in Chapter 5.

Example: Treedoc CRDTs were first introduced by Preguiça et al. when they proposed Treedoc [PMSL09], an operation-based CRDT for collaborative editing. Consider a shared document consisting of a sequence of *atoms*. An atom is the smallest unit of content supported by the editor, e.g. a character of text. The basic idea of Treedoc is to assign a globally unique position identifier *pos* to every atom. Position identifiers are totally ordered, such that the total order is consistent with the order of the atoms in the document. Additionally, the space of position identifiers is dense, i.e. for any position identifiers pos_1 and pos_2, one can create a new position identifier pos_{new} with the property $pos_1 < pos_{\mathsf{new}} < pos_2$. Treedoc allows two operations:

- $\mathsf{insert}(pos, atom)$: Inserts the new atom *atom* into the document at position *pos*.
- $\mathsf{delete}(pos)$: Removes the atom at position *pos*. For the operation to be valid, such an atom must exist in the state of the device that initiates the operation.

In Treedoc, the principle is to use paths in a binary tree as position identifiers. Figure 2.2 shows a possible representation of the string "abcde". For example, the path for character 'e' is "11". The order of atoms in the document is given by an infix order depth-first traversal of the tree. When inserting a new atom, one derives a new position identifier by creating a suitable descendant of the node to the left or right of the desired insertion position. However, this alone is not enough to guarantee uniqueness of the identifier, since more than one user can perform insertions concurrently. To solve this, Treedoc treats each tree node as a *major node* that can contain multiple *mini-nodes*. Each mini-node contains an atom and has a *disambiguator* attached. Disambiguators need to be ordered

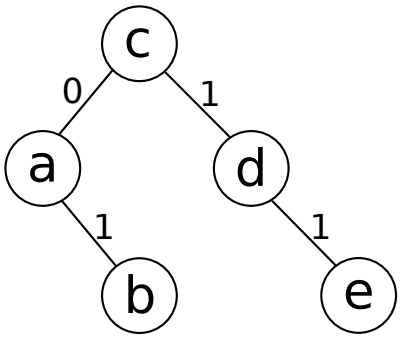


Figure 2.2: Possible tree representation of the string "abcde" as used in Treedoc.

and unique within a major node. When necessary, the paths used for position identifiers include the disambiguators. It is necessary only for the last node in the path, and if the path follows a child mini-node directly. If no disambiguator is given, a path element refers to a child of a major node.

Choosing disambiguators One can use the unique replica identifiers as the disambiguators. However, in this case, nodes must not be removed when atoms are deleted, since otherwise a node with the same path can be re-inserted later, breaking the global uniqueness of position identifiers. Instead, when a corresponding atom is deleted, the node is marked as a *tombstone*. Tombstones can be garbage-collected if it is clear that all replicas have seen the deletion and no further delete operation can refer to the old atom. However, if devices often go offline permanently without notice, such a garbage-collection scheme is impractical.

To avoid tombstones, one can instead use globally unique disambiguators, such as $(ctr, replicaID)$ pairs, where *replicaID* is a unique identifier of a replica, and *ctr* is a per-replica counter. In this case, leaf mini-nodes can be discarded immediately when the corresponding atom is deleted. Non-leaf nodes can be discarded when all its children have been discarded. Figure 2.3 shows an example representing the string "abcdxye" using $(ctr, replicaID)$ disambiguators.

Example: Replicated growable array (RGA) A replicated growable array (RGA) is another CRDT for lists, proposed by Roh et al. [RJKL11]. It allows insertions, updates, and deletions of elements. Each operation has an associated unique timestamp. Timestamps are totally ordered, and this ordering needs to be consistent with the causal order of operations. For example, given a vector clock $\overrightarrow{v}_O$ for an operation O, one can use a pair $(sumv, sid)$, where $sumv = \sum_i \overrightarrow{v}_O[i]$, and *sid* is a unique site identifier. A position in the list is identified by the timestamp of the operation that inserted it.

An RGA defines three (remote) operations:

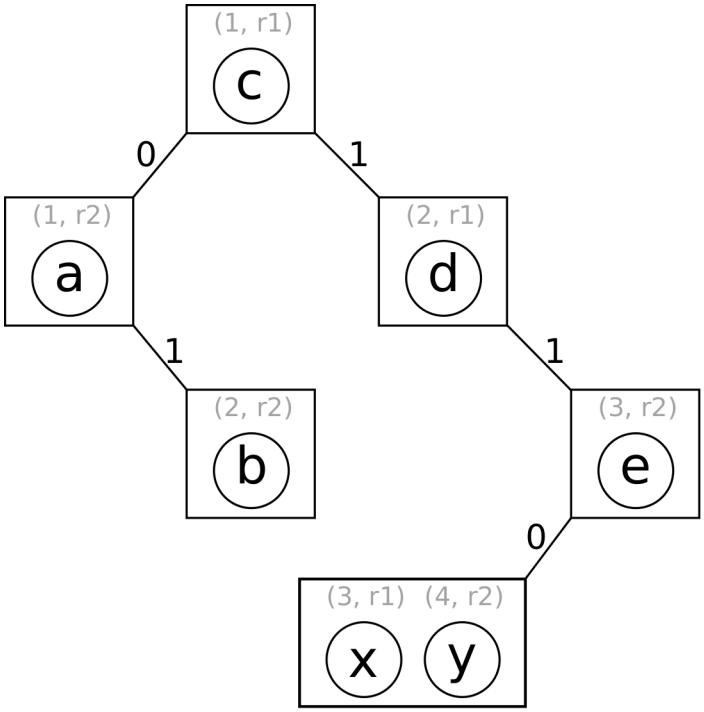


Figure 2.3: Possible representation of the string "abcdxye" in Treedoc using globally unique disambiguators. The atoms 'x' and 'y' have been inserted concurrently at the same position of the tree; the disambiguators are used to create unique position identifiers and a consistent ordering of these atoms.

- insert_after(*timestamp*, *atom*): Inserts the new element *atom* into the document directly after the position *timestamp*.
- update(*timestamp*, *atom*): Changes the value of the element at the position *timestamp* to *atom*.
- delete(*timestamp*): Removes the element at the position *timestamp*.

To be valid, all operations require that the corresponding position exists at the time the operation is initiated. However, an insertion may refer to an adjacent position that is deleted concurrently, and since such an insertion may arrive at a replica after the deletion, deleted list nodes need to kept as tombstones.

RGA achieves commutativity of concurrent operations by using the ordering of timestamps. Concurrent insertion operations are treated as if they have happened in increasing order of their timestamps. If multiple updates happen concurrently for a position, the update with the highest timestamp is effective. If update and delete operations happen concurrently, the deletion takes effect.

These rules ensure that there is a uniquely defined state for each set of operations, independent of the order concurrent operations are received in. Consider two concurrent operations O_a, O_b, where O_b has a higher timestamp than O_a. If a replica receives O_a after

it has already applied O_b, it could undo O_b and then apply both operations in the right order. However, for better efficiency, RGA is designed such that it never requires undoing of operations; instead concurrent operations can be applied directly in any order.

For example, consider a document containing $[a, b, c]$. Now two replicas concurrently insert x and y, respectively, after a: $O_x = \mathsf{insert\_after}(\mathit{timestamp}_a, x)$, $O_y = \mathsf{insert\_after}(\mathit{timestamp}_a, y)$. Let $\mathit{timestamp}_x$ and $\mathit{timestamp}_y$, respectively, be the timestamps of the operations, and suppose $\mathit{timestamp}_x < \mathit{timestamp}_y$. Thus, the correct final state is $[a, y, x, b, c]$. Each replica keeps a linked list of nodes storing a 3-tuple $(\mathit{timestamp}, \mathit{lastupdate}, \mathit{atom})$, where *timestamp* is the timestamp of the operation that inserted the element, *lastupdate* is the timestamp of the operation that last updated the element, and *atom* is the content. A replica that applies O_y first is in state $[a, y, b, c]$, so naïvely applying O_x by inserting x after a would result in a wrong state. Instead, when applying an insert, the algorithm skips over any elements right of a that have a higher *timestamp*. Since $\mathit{timestamp}_y > \mathit{timestamp}_x$, and $\mathit{timestamp}_x > \mathit{timestamp}_b$ (since the operation that inserted b happened before O_x), x ends up in the correct place.

A deletion marks an element as deleted (if it has not already been marked). Updates are only applied if the element is not marked as deleted, and no update with a higher timestamp has been applied previously.

2.1.2.3 Drawbacks of CRDTs compared to OT

As explained above, CRDTs have a number of advantages over OT. However, they also come with some costs. Unlike OT algorithms, CRDTs for collaborative editing assign a unique immutable identifier to each atom, which creates additional metadata overhead and therefore space and communication complexity. CRDTs for collaborative editing can be grouped into two categories: tombstone-based CRDTs such as RGA, and variable-size identifier CRDTs such as Treedoc or Logoot. In tombstone-based approaches, identifier information must be retained in the document in the form of tombstones, even if the corresponding portions of the document are deleted by the user. Tombstones can only be discarded after a garbage collection process that requires distributed consensus and is rarely implemented in practice. On the other hand, CRDTs with variable-sized identifiers do not require tombstones to be retained, but they have the downside that identifiers grow quickly. For example, consider Treedoc and suppose that a user writes a piece of text from left to right. This causes a highly unbalanced tree that resembles a linked list and the identifiers to grow linearly with the number of edits. This growth can be somewhat reduced using an optimised allocation strategy such as in LSEQ, but the identifiers in practice remain substantially larger than a simple index.

2.1.3 Version control systems

For collaborative editing of source code and other files that are part of software development projects, version control systems are widely used. While the collaborative editing techniques above (OT and CRDTs) are more suitable for individual files, version control systems are designed to handle large numbers of files; however, modern version control systems also allow concurrent editing of individual files.

The first popular version control system was called Source Code Control System (SCCS), developed in 1972 by Marc Rochkind at Bell Labs. Users had to acquire an exclusive lock on a file before working on it, and thus could not edit the same file concurrently.

The second generation of version control systems, including CVS (1985) and Subversion (2000), used a client/server model. The server keeps a repository holding the authoritative copy of all files and metadata; clients *check out* files from the repository and keep a local copy that they work on. The client needs to *commit* changes to the server to share them with other collaborators. When committing, the server checks whether another client has committed changes since the last checkout. If so, the client trying to commit first needs to *merge* the remote changes into their local copy. Merging can often be done automatically, but in some cases needs to be performed manually by the user, e.g. if the same source code line was modified by both.

More recently, distributed version control systems have started to be widely used. Git (2005) and the simultaneously developed Mercurial are two popular examples. Distributed version control systems follow a peer-to-peer approach, where every peer holds a complete copy of a repository. This allows peers to perform commits and other operations locally, making them faster and allowing offline operation. Commits can be exchanged directly between peers in the form of patches. Concurrent changes are again merged automatically or manually. In practice, it is common to use a server as a specialised peer that holds the authoritative copy.

Distributed version control systems like Git have some similarities with operation-based CRDTs: they are both decentralised; and a basic commit in Git is essentially a set of deletion and insertion operations. An important difference is that in OT and CRDTs, there are no manual merges, since the same set of operations always has to result in the same final state. This makes OT and CRDTs better suited for real-time collaboration, where requiring manual merges would be disruptive, and the user interface can usually be designed to allow users to easily spot if another user is editing the same part of the document concurrently. On the other hand, it makes them less well suited for off-line collaboration on documents like source code. Using OT or CRDTs, if two users concurrently edit the same line of a source code file (assuming character-based atoms), the result will be a combination of both changes, which can lead to a nonsensical result. Some mechanism for conflict detection and manual resolution is needed for non-real-time collaborative systems.

2.2 Secure collaboration

Numerous approaches for real-time collaborative editing have been proposed that reduce the amount of trust that clients need to put into a central server, e.g. SPORC [FZFF10], SECRET [FMMS17], or a system proposed by Huang and Evans [HE11]. These systems are usually primarily concerned with confidentiality of the shared document. Integrity and consistency is considered to varying degrees; some systems use authenticated encryption and/or hash chaining.

In terms of communication patterns, designs can be split into centralised and peer-to-peer architectures. Currently, many existing systems that aim for secure collaboration require a central server because they rely on OT algorithms that need a total ordering of operations.

Systems that offer confidentiality can be categorized into document-based encryption, and operation-based encryption.

Document-based encryption

In document-based encryption, the document plain-text is encrypted, and a collaboration algorithm such as OT is used on the ciphertext. An advantage of using the document-based approach is that it can be used on top of an existing OT system. However, using commonly used block cipher modes of operation, such as GCM or CBC, adding a single character can cause huge changes in the ciphertext; naively applying this approach is inefficient and does not work if users concurrently perform edits. To address the first problem, Huang and Evans propose using incremental encryption [HE11] using the uncommon RPC mode of operation [BGG94]. To address both problems somewhat, Felsch et al. propose a system called SECRET [FMMS17], where they split the document into small parts and encrypt them individually using GCM mode.

Operation-based encryption

In contrast to the document-based approach, operation-based encryption encrypts individual operations or parts of them. In this category, Feldman et al. propose a collaboration system called SPORC [FZFF10] based on OT, where clients encrypt operations using a symmetric key, and sign them using the individual signing keys. It requires a server only to totally order all operations. The system uses hash chains to ensure the integrity of the editing history, and to achieve *fork* consistency* [LM07]. A malicious server can partition the clients into two or more groups and only show them edits from their group starting from some point. However, once two clients from different groups are able to communicate, they will detect such a fork. SPORC also has support for snapshots. However, it does not offer a way for a new collaborator to verify the integrity of a snapshot other than verifying

that the user who signed the snapshot had permission to access the document when they signed it.

Clear et al. use a different approach that, unlike SPORC, does not require a custom server implementation and can be run on top of an existing OT system, in their case Google Docs. Here, insert operations are transformed such that the content is encrypted. The server can still observe the type of operations and their length, and can perform transformations on them.

The protocol proposed in Chapter 5 falls into the operation-based encryption category.

2.3 Cryptographic accumulators

The protocol presented in Chapter 5 uses cryptographic accumulators as an efficient way for devices to attest their current state, while allowing other devices to prove certain statements about that state using the accumulator, without revealing the state itself.

A cryptographic *accumulator* allows a finite set $\mathcal{X}$ to be accumulated and represented by a single, constant-sized value, $acc_{\mathcal{X}}$. For every element $s \in \mathcal{X}$, one can efficiently compute a *witness* $w_{s \in \mathcal{X}}$ that can be presented to prove the *membership* of s with regards to $acc_{\mathcal{X}}$, i.e. proving that s is part of the set accumulated in $acc_{\mathcal{X}}$. However, it is computationally infeasible to compute a witness for an element $x \notin \mathcal{X}$ (*collision-freeness*).

Some accumulator schemes allow efficient addition and deletion of elements to an existing accumulator. These are called *dynamic* accumulators. Accumulator schemes may also allow proofs of non-membership, i.e. a proof that a certain element has not been accumulated. Such an accumulator scheme is called *universal*.

Another desirable property of accumulators is *indistinguishability*. Informally, an accumulator is computationally/unconditionally indistinguishable if it is infeasible/impossible for an adversary to gain any information about the accumulated set from the accumulator value. A precise definition is given in [DHS15]. For some accumulator schemes, indistinguishability can be achieved by accumulating an additional, random value. This has the caveat that it weakens collision-resistance by allowing membership proofs for the random value.

Known accumulator schemes often require generating a key pair $(\mathsf{sk}, \mathsf{pk})$, where the secret key sk acts as a trapdoor. Knowledge of the trapdoor allows breaking the collision-freeness property. For these schemes, depending on the application, a trusted setup may be required, where a trusted third party generates the key pair and discards the trapdoor. For dynamic accumulators, the secret key may be required for efficient additions and/or deletions.

Cryptographic accumulators have been been proposed based on RSA [BP97, BD93], bilinear maps [Ngu05, CKS09], Merkle trees [CHKO12], and vector commitments [CF13].

The following explains the RSA accumulator in more detail, which is also used in Chapter 5. Section 2.3.2 briefly explains an alternative way to construct accumulators, based on bilinear maps.

2.3.1 RSA accumulator

RSA accumulators [BD93, BP97] are based on the RSA assumption [RSA78]. An RSA accumulator requires an RSA secret key consisting of two safe primes[1] p and q, and a base value x that is drawn randomly from the cyclic group of quadratic residues modulo N, where $N = pq$ is the RSA modulus [DHS15]. In the elementary form of the accumulator, the accumulator value is calculated as:

$$acc_{\mathcal{X}} = x^{\prod_{a \in \mathcal{X}} a} \mod N. \tag{2.1}$$

Due to the multiplications in the exponent, the elements to be accumulated are restricted to prime numbers for collision-freeness. Additionally, it requires a stronger variant of the RSA assumption, the *strong RSA assumption* [BP97].

While anyone with knowledge of x and N can accumulate values, it is more efficient with knowledge of the secret key. Accumulating k values requires k modular exponentiations without knowledge of the factorization of N, but can be reduced to a single modular exponentiation and $k - 1$ modular multiplications if the factorization is known. The exponent is computed by multiplying all elements modulo $\varphi(N) = (p - 1)(q - 1)$.

Generalisation to allow all integers

To remove the restriction to prime numbers, Barić and Pfitzmann proposed a variant of the accumulator that uses *prime representatives* [BP97]. They constructed prime representatives as follows:

Let $\Omega(a)$ be a random oracle that returns a random number r for each new input, and returns the same number r if asked again for the same input. The prime representative $h(a)$ for an element a is computed as $h(a) = 2^l\Omega(a) + d$, where l is suitably large, and d is an l-bit number chosen such that $h(a)$ is a prime. In other words, one appends l bits to $\Omega(a)$ such that the result becomes prime. Using this variant based on a random oracle also removes the dependency on the strong RSA assumption, relying on the normal RSA assumption instead [BP97]. In practice, the random oracle can be replaced by a secure hash function. Collision-freeness of the accumulator then also depends on the collision-freeness of the hash function. For better readability, we omit this hash function in the remainder of this section.

[1] A prime p is *safe* if $p = 2p' + 1$, where p' is an odd prime [BD93].

Membership proofs

Let $b \in \mathcal{X}$. To show that b is accumulated in $acc_{\mathcal{X}}$, one presents a witness $w_{b \in \mathcal{X}}$ computed as:

$$w_{b \in \mathcal{X}} = x^{\prod_{a \in \mathcal{X} - b} a} \mod N. \tag{2.2}$$

Thus, a witness for an element is equal to the accumulator value for all the remaining accumulated elements. To verify the correctness of the witness, one checks:

$$w_{b \in \mathcal{X}}^{b} \mod N = acc_{\mathcal{X}}. \tag{2.3}$$

In addition to memberships proofs for individual elements, RSA accumulators allow batch membership proofs, i.e. one can provide a single witness for a subset $S \subseteq \mathcal{X}$. The witness is again computed by accumulating all remaining elements, and verification works accordingly:

$$w_{S \subseteq \mathcal{X}} = x^{\prod_{a \in \mathcal{X} \setminus S} a} \mod N, \tag{2.4}$$

$$(w_{S \subseteq \mathcal{X}})^{\prod_{a \in S} a} \mod N \stackrel{?}{=} acc_{\mathcal{X}}. \tag{2.5}$$

Note that computing a witness does not require knowledge of the secret key; it requires only x, N, and the accumulated set $\mathcal{X}$. On the other hand, anyone in possession of the secret key can efficiently forge membership proofs for arbitrary non-accumulated elements. For any $c < N$, one can compute $w_{c \in \mathcal{X}} = acc_{\mathcal{X}}^{\frac{1}{c}} \mod N$, which can be done efficiently if the factorization of N is known.

Updates

The RSA accumulator is a dynamic accumulator scheme, allowing efficient additions and deletions to an existing accumulator instance. Adding an element is straightforward:

$$acc_{\mathcal{X} \cup \{e\}} = acc_{\mathcal{X}}^{e} \mod N. \tag{2.6}$$

Removing an element e from the accumulator requires calculating the e-th root modulo N. This is essentially the same as performing an RSA decryption, and thus assumed to be hard without knowledge of the factorization of N. If the factorization is known, it can be done efficiently by raising to the multiplicative inverse of e modulo $\lambda(N) = \mathsf{lcm}(p-1, q-1)$. Here, $\lambda(.)$ is the Carmichael function, which gives the smallest positive integer m such that $a^m \equiv 1 \pmod N$ for every positive integer a less than N that is coprime to N. The inverse can be computed using the extended Euclidean algorithm to find t such that $t \cdot e \equiv 1$

(mod $\lambda(N)$), which is equivalent to solving the following linear diophantine equation:

$$t \cdot e = u\,\lambda(N) + 1. \tag{2.7}$$

Then, the e-th root modulo N can be computed by raising to the power of t:

$$(z^e)^t \equiv z \cdot \left(z^{\lambda(N)}\right)^u \equiv z \cdot 1^u \equiv z \pmod{N}, \tag{2.8}$$

and hence the accumulator can be updated as follows:

$$acc_{\mathcal{X}\setminus\{e\}} = acc_{\mathcal{X}}^t \mod N. \tag{2.9}$$

To update an existing witness w after an element e was added to the accumulator, w is raised to the e-th power modulo N. If e was removed from the accumulator, a witness $w_{y\in\mathcal{X}}$ for an element $y \neq e$ can be updated by finding a and b such that $ae + by = 1$. The updated witness is calculated as:

$$w_{y\in\mathcal{X}\cup\{e\}} = w_{y\in\mathcal{X}}^b\, acc_{\mathcal{X}\cup\{e\}}^a \mod N \text{ [CL02]}. \tag{2.10}$$

Indistinguishability

RSA accumulators can achieve indistinguishability by using the transformation described earlier, i.e. accumulating an additional (secret) random value. Equivalently, one can choose a random base value [dMLP+12, DHS15].

2.3.2 Accumulators based on bilinear maps

Accumulators in this class are based on the Strong Diffie-Hellman (SDH) assumption [BB04], or the Diffie-Hellman Exponent (DHE) assumption [CKS09]. Accumulators based on the SDH assumption were proposed by Ngyuen [Ngu05]. Let $\mathbb{G}$ be a cyclic additive prime order group generated by g, and let $\mathbb{G}_{\mathbb{M}}$ be a cyclic multiplicative group of the same order. Let $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_{\mathbb{M}}$ be a bilinear map. The accumulator has an upper bound t for the number of accumulated elements. The accumulator requires a secret key s, and the public key is $(g, g^s, g^{s^2}, ..., g^{s^t}, u)$, where u is randomly chosen from $\mathbb{Z}^*$. The accumulator value for a set $\mathcal{X}$ of size at most t is calculated as $acc_{\mathcal{X}} = g^{u\prod_{a\in\mathcal{X}}(a+s)}$, and the witness for an element $b \in \mathcal{X}$ is computed as $w_{b\in\mathcal{X}} = g^{u\prod_{a\in\mathcal{X}\setminus\{b\}}(a+s)}$. The accumulator value can also be evaluated without the secret key by expanding the polynomial $h(x) = \prod_{a\in\mathcal{X}}(a+x) = \tilde{a_0}+\tilde{a_1}x+\tilde{a_2}x^2+...+\tilde{a_t}x^t$, and evaluating $g^{h(s)}$ using the public key. The witnesses can be evaluated similarly.

2.4 Redactable signature schemes

As mentioned in the previous section, in Chapter 5, we use signed cryptographic accumulators as a way for devices to attest (sign) their current state. When creating an authenticated snapshot, a device essentially includes these signatures from other devices. Since some devices may have not received the most recent edits (because they have been offline), we make use of the concept of redactable signatures to redact deleted atoms.

A redactable signature scheme (RSS) [SBZ01, JMSW02] allows a party without knowledge of the secret signing key to remove parts of a signed message and to update the signature, such that the redacted message can be verified using the updated signature and the public key of the original signer. This gives a useful privacy-preserving property, e.g. for redacting medical histories of patients. The redactability property can be trivially achieved by signing each atomic submessage of the original message using a standard digital signature scheme, and throwing away the signatures of any redacted parts. This approach is not efficient in terms of computational and communication costs; a desirable scheme should improve in at least one of these dimensions.

One example RSS is the one described by Johnson et al., based on the homomorphic properties of RSA [JMSW02]. The scheme works similar to RSA accumulators (Section 2.3.1). Again, the signing party chooses an RSA key pair $N = pq$ and a base x. Let $h(.)$ be a hash function returning odd primes like the one described in Section 2.3.1, extended to sets such that $h(S) = \prod_{a \in S} h(a)$. To sign a set S, the signer computes $y = x^{\frac{1}{h(S)}} \mod n$. To verify the signature, one checks that $y^{h(S)} = g$. For $U \subseteq S$, the signature for the redacted subset $S \backslash U$ can be obtained by computing $y' = y^{h(U)}$. Another way of getting an RSS is to add all atomic submessages to an accumulator, sign the accumulator using any digital signature scheme, and provide witnesses for all submessages. To redact parts of a message, one removes the corresponding witnesses [DPSS16, PSP12]. For a message containing m submessages, this in general requires $\Theta(m)$ witnesses as part of the signature. However, if the accumulator scheme has certain properties such as allowing batch membership verification, the signature size can be reduced to $\mathcal{O}(1)$ [DPSS16]. This is possible, for example, with the RSA accumulator, because it supports witnesses for subsets of the accumulated set, and such witnesses can be computed without knowledge of the secret signing key.

2.5 Merkle trees

Another cryptographic tool used in Chapter 5 is a Merkle tree. We employ them to allow devices to check consistency of signatures from different devices that are part of an authenticated snapshot.

A Merkle tree (sometimes referred to as hash tree) is a tree, where each leaf node is labelled with the cryptographic hash of a data block, and each non-leaf node is labelled with the hash of its children's hashes [Mer88]. The hash associated with the tree root is called the *root hash*. Merkle trees allow efficient verification of the validity of individual data blocks. Due to the collision-resistance of the underlying hash function, verifying any data block at the leaf only requires a trusted root hash, and the hashes of any siblings of nodes along the path to the root, which do not need to be trusted. This also allows proving that a certain value is contained in the Merkle tree. This property makes Merkle trees suitable for use as cryptographic accumulators [CHKO12].

Furthermore, Merkle trees allow running a tamper-evident append-only log on an untrusted server [CW09], as in Certificate Transparency [LLK13]. In addition to the membership proofs above, such logs use a predetermined way of constructing the tree, and *consistency proofs*, which, given the root hashes (commitments) of two states of the same log, provide an efficient proof that the earlier state is a prefix of the later one. A consistency proof consists of a subset of nodes from the later tree that allows verifying the root hash of this tree, and – using a subset of the same nodes – the root hash of the earlier tree.

2.6 Anonymous communication

A common way to define anonymity in the context of communication systems is as "the state of being not identifiable within a set of subjects, the *anonymity set*." [PK01] Therefore, anonymity is a relative concept; it depends on the size of the anonymity set and on any additional information held by the adversary that could help them identify the subject. In anonymous communication, one differentiates between anonymity of the sender (*sender anonymity*), and the receiver (*receiver anonymity*) of a message, and the respective anonymity sets. More generally, anonymity systems aim for *unlinkability* of some sorts. Unlinkability "of two or more items (e.g. subjects, messages, events, actions, ...) means that within this system, these items are no more or no less related than they are related concerning the a-priori knowledge." [PK01] This can mean, for example, the unlinkability between a sender and a message, between sender and recipient, or between two messages. A stronger concept than anonymity is *unobservability*, which in the context of anonymous communication means that the adversary does not find out whether a particular sender sends, or a particular receiver receives a message at all.

Most modern anonymous communication systems rely on *relays*, which forward messages or packets to other relays or to the intended recipient. An important concept introduced by Chaum are *mixes* [Cha81]. A mix collects a number of encrypted messages, decrypts them, and later forwards them in a batch. The idea is that an adversary who can observe

the incoming and outgoing network links should not be able to link incoming and outgoing messages. To avoid having to trust a single mix, in mix networks, a sender chooses a sequence of mixes, encrypts the message in layers with the public key of each of the chosen mixes, and has the messages routed through all of those mixes. Mixmaster [MCPS04], Babel [GT96], and Mixminion [DDM03] are systems that use different variants of this design. The anonymity in these systems comes at the cost of high latencies, since to prevent traffic analysis, mixes need to delay forwarding messages.

To support low-latency communication, onion routing [GRS99] was developed. In onion routing, packets are encrypted in layers and sent through a number of relays, similar to mix networks. Unlike mix networks, onion routing is circuit-based however. The most popular low-latency anonymity system, Tor [DMS04], is based on this principle. Section 2.6.1 explains Tor in more detail. Since onion-routing is aimed at low-latency applications, such as web browsing, the mixing performed by relays is minimal, making it susceptible to traffic analysis attacks and therefore not resistant against a global passive network adversary. For example, an adversary can link streams going into and coming out of the Tor network if they can observe both network links.

More recently, Piotrowska et al. proposed Loopix [PHE+17], an anonymity system that is based on mix networks, but uses a combination of mechanisms to allow lower latencies than traditional mix networks, while still being resistant against a global passive adversary and some active attacks. To achieve this, it uses several types of cover traffic, and random forwarding delays at the mixes (from an exponential distribution). Furthermore, every client is connected to a provider, which provides offline storage for received messages. Message delays in Loopix are typically in the order of seconds. This combination of properties could make Loopix an attractive building block for future collaborative applications resistant against a global passive adversary that aim for a relatively low latency, but where a few seconds delay and the cost of the required cover traffic are acceptable.

In Chapter 4, we explore the use of Tor as a privacy-preserving communication channel for collaborative applications. The remainder of this section describes Tor in more detail to give the reader the necessary background for the analysis in Chapter 4.

2.6.1 Tor

The Tor [DMS04] network is composed of *clients*, which generate and consume traffic, including smartphones, laptops, or desktops; and servers (relays), which forward traffic to other relays and make connections to the public Internet on behalf of clients. To use the Tor network, clients download the latest *network status document* on average approximately every 90 minutes, which lists information about currently (December 2018) around 6 400 Tor relays available worldwide [Tora]. The network status document is managed by a small number of more trusted servers, called *directory authorities*, who vote on a consensus

of its contents once an hour. The directory authorities publish additional, relatively static, information on relays in *relay descriptors* every 18 hours. After downloading the network status document, clients download any relay descriptors mentioned in that document that the client does not already have. The client also downloads certificates of authorities where it does not already have a current one.

Clients use relays to build *circuits* through a sequence of (typically three) relays. Such circuits support an overlay network between the client and the final public Internet service required. The client applies layers of encryption in such a way that none of the relays, nor the final Internet service, is able to determine which devices on the Tor network are connecting to which Internet services. Because circuit construction takes time, clients proactively build circuits in anticipation of any requirement for data connectivity; circuits can also support multiple concurrent TCP streams. Tor clients periodically send keep-alive messages on idle open connections to prevent the connection from expiring at any intermediate routers. The default interval between keep-alive messages is currently 5 minutes. To improve the privacy properties of Tor, circuits are (at least partially) rebuilt every 10 minutes.

2.6.2 Tor hidden services

In addition to supporting clients connecting to services such as websites on the public Internet, Tor also allows Tor clients to publish *hidden services*[2]. A hidden service is identified by an *onion address*; in version 2, the onion address consists of the first 16 characters of a base32-encoded SHA1 hash of a public key generated by the client. In 2017, the Tor team announced version 3 of hidden services, where the onion address is a 56-character string containing the base32-encoded version of the concatenation of the full public key, a checksum, and a version field. The relevant research described in this dissertation was conducted before, and so version 2 of the protocol is described here. The principles remain the same in version 3, but it uses more recent cryptographic algorithms and other mechanisms to improve security. Onion addresses are long-lived identifiers, distributed through an out-of-band mechanism between parties who wish to communicate. For example, Facebook offers a Tor hidden service at `https://facebookcorewwwi.onion/`.[3] An onion address allows Tor clients to establish circuits with the hidden service using the Tor *rendezvous* protocol [Torb, Torc], and therefore transfer data to and from the service over the Tor network. The design of Tor hidden services prevents any single relay from learning the IP address associated with an onion address, therefore providing anonymity to

[2]Hidden services are now officially called "onion services"

[3]Note: Facebook have spent considerable computational resource to final a public key whose base32-encoded SHA1 is memorable.

both a hidden service provider and its clients. The rendezvous protocol is described below, where we assume that Bob wants to run a hidden service and Alice wants to connect to it.

Bob creates a hidden service

1. Bob asks his Tor client to create a new hidden service. This generates a public-private key pair for the service. The *public key* of the service identifies the service and is used to generate an onion address.

2. Bob shares his onion address with Alice via an out-of-band mechanism.

Bob runs a hidden service

3. Bob's Tor client chooses a small number of (typically three) relays as *introduction points*. Bob then establishes a circuit to each introduction point and sends a single-use public key, or *service key*, and signs a message to prove he is the owner of this public key.[4] Bob's Tor client must keep the circuits to the introduction points open while the service is running to receive connection requests from new clients.

4. Bob's Tor client generates a *service descriptor* containing the public key, the service key, and the introduction points. The service descriptor is uploaded to a few (currently six) hidden service directories, chosen based on the descriptor ID, which is a hash of the service's public key, the current date and time, and other deterministic data. Bob's Tor client publishes a new descriptor once an hour, or whenever its content changes.

Alice connects to Bob's hidden service

5. Alice's Tor client determines the set of hidden service directories responsible for Bob's key using his onion address and the current time, and retrieves Bob's service descriptor from one of them.

6. Alice's Tor client establishes a *rendezvous* point. It does so by randomly choosing a Tor relay, building a circuit to it, and asking it to act as a rendezvous point, specifying a randomly chosen 20 byte *rendezvous cookie*.

7. Alice's Tor client connects to one of Bob's introduction points and requests an introduction to Bob by providing a hash of Bob's service key. Alice also sends a rendezvous request, including the address of the rendezvous point, the rendezvous cookie, and the first part of a Diffie-Hellman key exchange, all encrypted under Bob's temporary service key.

[4]Earlier versions used the public key of the hidden service instead of a single-use service key, but this allowed the introduction point to monitor Bob's activity.

8. The introduction point forwards the rendezvous request to Bob. Bob checks the request is valid and not a replay.

9. Bob's Tor client creates a new circuit to the rendezvous point chosen by Alice and asks the rendezvous relay to complete a circuit to Alice. Bob's request contains the rendezvous cookie, the second part of the Diffie-Hellman exchange, and a handshake digest. The rendezvous point forwards the latter two to Alice's Tor client. Alice's Tor client checks that the handshake is valid, and both sides derive a new set of keys. A new circuit is now established between Bob and Alice.

10. Alice can now establish one or more TCP connections over her circuit with Bob.

2.7 Summary

This chapter briefly summarised the background required for the rest of the dissertation. It introduced the two dominant approaches for real-time collaborative editing: OT and CRDTs. The majority of OT algorithms rely on a central server, making them badly suited for a decentralized architecture and end-to-end encryption. For this reason, the work described here is based on CRDTs, which are decentralised by design. The chapter also considered version control systems, which are more suitable for larger projects and non-real-time editing, and existing approaches to secure collaborative editing.

The remainder of the chapter summarized a number of computer security and cryptography concepts needed to achieve the privacy and integrity properties of the protocol presented in Chapter 5. Cryptographic accumulators, redactable signatures, and Merkle trees were briefly explained. Lastly, the most important approaches for anonymous communication systems were summarised. Tor was described in more detail to provide the necessary background for Chapter 4, which explores the costs of running Tor hidden services on smartphones.

CHAPTER 3

Privacy-preserving collaborative applications

As discussed in the Introduction, currently deployed systems for collaboration, such as Google Docs, require users to fully trust the service provider with their data and metadata. Since this is often undesirable from a privacy and security standpoint, in this dissertation we explore systems with better privacy properties.

We envision a collaboration software system that can be used by any number of users. Users can form groups of collaborators, where every collaborator can view and edit a shared document or other data set. Every user can have one or more devices, and every participating device stores a copy (replica) of the shared document or data set. The system should support end-to-end encryption between devices to protect confidentiality of the shared data set, and it should protect metadata about the shared data set, such as the document name, and about users, such as identity, location, and IP address.

For simplicity, we usually consider individual devices instead of users, since in most cases, devices of the same user can be treated the same as devices from different users. We consider both applications where data is shared between multiple users, as well as applications where data is synchronised between multiple devices owned by the same person.

A data set consists of indivisible elements (atoms). In a collaborative editing application, an atom is typically a character, a word, a line, or a paragraph. In a photo sharing application, individual photos could be atoms. Every collaborating device can add to or delete atoms from the data set. Edit operations can be performed concurrently on multiple devices, and devices can also perform modifications while they are only connected to a subset of collaborators, or offline. No consensus between devices is required; instead, the system provides eventual consistency. We assume a design where every collaborator can add new devices to the group. However, in principle, users could require a corresponding

permission to do so. Devices may also be removed and have their membership revoked; however we do not consider revocation in this dissertation.

The prime example of the kind of collaborative application we consider is a collaborative text editor, where users share a text document, and every collaborator can add or delete text. In Chapter 5, we present a protocol for such a collaborative text editor. However, the principles discussed apply to a wider range of collaborative applications. Other applications that can use the same principles and protocols include shared or synchronized to-do lists, note-taking applications, calendars, address books, password managers, or photo albums.

3.1 Design goals

We divide our design goals into three groups: functionality, integrity properties, and privacy properties. For integrity and privacy goals, we define a threat model in the next section (Section 3.2).

Functionality: The design must allow real-time collaborative applications, where users that open a shared document simultaneously can see edits by themselves and by other collaborators with minimal delay. Users should continue to be able to edit the document if they are offline. Similarly, users should be able to continue editing and collaborating with other users that are reachable, even if any number of other collaborators are offline or partitioned. The design should be able to support snapshots, i.e. new collaborators should be able to start from a compacted state of the document and not require the past edit history. Lastly, the system should not incur large costs in terms of computation, memory, storage, or communication, even if a document grows relatively large. We aim for the system to be usable on resource-constrained devices such as smartphones.

Integrity: The system needs to have a way to ensure that only authorized users are able to perform edits of the shared document. All edits should be attributable to the originator. Users should also be able to identify the author of parts of the document that were added before they joined. If any two honest devices have received the same set of messages, they must have the same view of the document. More generally, honest devices must always have consistent views of the document. Edit operations are causally ordered, and after a device processes an operation, its state must reflect this operation, all operations that have happened before this operation, and possibly some concurrent operations. Any inconsistencies or forks between edit histories of honest devices should be promptly detected if these devices communicate. Devices should be able to resolve any forks they detect.

Privacy: The contents of a shared document must be visible only to authorized collaborators. Non-collaborators should not be able to observe metadata about the collaboration, such as a document title or the identities of the collaborators. Furthermore,

new collaborators should not be able to see the history of edits that happened before they joined.

Assumptions

We assume devices are recent personal computing devices like desktop computers, laptops, tablets, or smartphones. Such devices can frequently be offline for largely varying time periods, and may stop participating without notice, for example if the device is lost or breaks. To exchange messages, collaborating devices need to have a way of communicating either directly, e.g. via a local network, Bluetooth, or peer to peer over the Internet, or indirectly via a server or other collaborators. Thus, we assume that a connected graph of communication links exists between all collaborating devices. However, not all devices need to be connected simultaneously and they may be partitioned temporarily. While group size is not limited, we assume that groups of collaborators remain relatively small and accordingly we optimize performance for smaller group sizes.

3.2 Threat model

We consider the following types of adversaries:

Global network adversary. A global network attacker can observe, and arbitrarily create, modify, delay, or drop network traffic between collaborators.

Non-global network adversary. A non-global network attacker can perform the same actions, but can only control traffic on a small fraction of the network.

Malicious collaborator. A malicious collaborator has the same abilities as an honest collaborator, but may deviate arbitrarily from the protocol. He may further create any number of additional collaborators and collude with them.

For the anonymity of collaborators, we aim for resistance against a non-global network adversary, as in real-time anonymity systems such as Tor. Investigating the use of anonymous communication systems with a stronger threat model, such as Loopix [PHE+17], is an interesting direction for future work. For all other privacy and integrity properties, we aim for resistance against a global adversary, and, where relevant, a malicious collaborator.

Adversaries might attempt to achieve the following goals:

Break confidentiality. Network adversaries may try to break confidentiality, to obtain plain text contents of a shared document.

Link collaborators. Network adversaries may try to find out whether a particular person or device is collaborating on a particular document or with a particular other device.

Break consistency. Any adversary may try to cause honest devices to have inconsistent views of a shared document.

To simplify analysis and to focus on particular aspects of the problem, we make a number of assumptions.

- Adversaries are not able to break standard cryptographic primitives such as encryption, digital signatures, cryptographic hashes, and accumulators.
- Every collaborating device is able to verify the authenticity of other devices' public keys, for example through a public key infrastructure (PKI).
- Devices do not have access to the private keys of other honest devices.
- To ensure progress, network adversaries cannot partition the network indefinitely.

3.3 Scope of this work

Essential components of the kind of collaborative application we consider include: a user interface; algorithms and data structures that can handle modifications to the shared state; a transport mechanism that allows collaborating devices to exchange messages that preserve both confidentiality of the contents as well as metadata privacy; a way to authenticate other users such as a PKI; and mechanisms to ensure the integrity and consistency of the data shared between correctly behaving devices, even in the presence of malfunctioning devices or malicious collaborators.

In this work, we focus on two of these aspects. In Chapter 4, we consider using the Tor network to exchange messages between devices in a privacy-preserving and decentralised way. Since, in Western countries, smartphones have become the most frequently used computing device that is commonly used for such applications – but also one of the most resource-constrained one – we evaluate the costs of using Tor as a peer-to-peer communication medium between smartphones. In Chapter 5, we present two protocols that ensure the integrity and consistency of the data set seen by different collaborating devices. The first protocol assumes a fixed set of collaborators or that new collaborators receive the full set of editing operations from the time a shared document or data set was created. The second only sends the current state of the data set to new collaborators, providing edit history privacy by withholding the complete history of operations. Chapter 6 evaluates the overhead and savings of the later protocol, and argues how it fulfils the integrity, consistency, and privacy properties.

CHAPTER 4

Tor hidden services on smartphones

To distribute changes to a shared document to other collaborators, devices need a way to communicate. In this chapter, we explore the costs of running a Tor hidden service on a device that allows the phone to receive messages from any other machine connected to the Tor network. If every collaborating device runs a Tor hidden service, devices can exchange messages directly over the Tor network, without relying on a central server.

To allow the research results to be applied more generally, we consider a push notification service that allows arbitrary small messages to be transferred to a smartphone via a Tor hidden service running on the phone. We consider the costs for the receiving device in terms of battery usage and data transferred over the network.

Because mobile devices tend to have limited allowances for cellular data, we explore in detail the cost in terms of data transferred over the cellular network per month on typical end-user devices. Since this cost largely depends on the patterns of connectivity of individual devices, we first build an empirical model of the costs of the different parts of the Tor protocol, and combine this model with data about connectivity patterns of devices from the Device Analyzer project.

Lastly, we explore a number of strategies that aim to reduce the cellular data usage of running a Tor hidden service.

4.1 Push notifications

Push notification services provide reliable, energy efficient, store-and-forward messaging between servers and clients. This mode of communication is sufficiently compelling for mobile devices that push notification services are integrated into operating systems. For example, Google Cloud Messaging (GCM) is embedded into Android through the Google

Play Services API.[1] GCM is also available as a library for developers of iOS apps and developers of extensions for the Chrome web browser. Consequently, push notification services are widely used by apps to support both device-to-device communication (e.g. sending and receiving messages between users of a social media app) as well as supporting information dissemination (e.g. news and sports score apps).

Push notifications provide app writers and client device owners with four advantages: first, if the client device is switched off, or temporarily disconnected from the Internet, the push notification service will store messages and deliver them when the device is next online; second, push notification software on the client initiates a single long-lived TCP connection from the client device to the service, avoiding issues with NAT and firewalls, as well as removing the need to poll servers periodically for updates; third, multiple messages destined for a variety of apps on a single client device can be coalesced temporally and multiplexed down a single TCP connection, saving battery life and improving performance; finally, an app server can achieve *service fan-out* by sending a single copy of a message to a push notification service and requesting that the message is delivered to many devices on a group or topic basis.

There are downsides to push notifications however. From a privacy perspective, a push notification service has the disadvantage that the service can see the sender and the recipient of every notification across a broad range of apps and thus may conduct surveillance and censorship. While data is encrypted between the app server and the notification service, and between the notification service and the handset, there is no requirement for it to be encrypted end-to-end. Therefore, app data can often be read by the push notification server. In addition, regardless of support for end-to-end encryption between an app server and a handset, metadata on which handsets use which apps, as well as the location of the user (e.g. via the handset's IP address), are revealed to the notification service.

In this chapter we explore the design space of more privacy-friendly push notification services based on Tor.

4.2 Push notifications over Tor

We consider three overall designs: use push notification services as deployed today; connect to a single push notification service via Tor; or run a separate push notification service per app and connect to each of these via Tor. In the latter two, connections via Tor could be made outbound from the phone to the service or a Tor hidden service running on a smartphone.

[1] GCM has been deprecated by Google in April 2018, and has been superseded by its successor, Firebase Cloud Messaging [Per18].

Connecting to a single push notification service may be more energy efficient than using one push notification service per app since separate messages from multiple app servers (likely destined for a variety of apps on the same handset) can be coalesced into a batch for delivery in a single Tor circuit. The downside is that the push notification service learns the app servers (and therefore apps) communicating with a single handset, although it does not necessarily know the identity or location of the handset if such communication is sent over Tor.

Running a hidden service on a smartphone does not, at first glance, appear to provide much benefit over the use of an outbound Tor connection to a push notification service. Importantly, however, hidden services allow app developers to avoid using a push notification service at all if the aim of the app is to share data between client devices.

Mobile devices typically sit behind a NAT or firewall. Thus, direct phone-to-phone communication is often difficult or impossible. If every device operates a Tor hidden service, direct communication between two smartphones is now possible, as an onion address is globally unique and accessible. The downside to this approach is that both the sending and receiving smartphone need to be online simultaneously for data to flow. This requires careful scheduling of smartphones to wake from low-power states and both devices to have network connectivity at the same time. An energy- and data-efficient solution is likely a prerequisite for mobile apps that use device-to-device communication, such as messaging or collaborative apps. We therefore focus on data and energy issues of Tor hidden services. We leave the issue of scheduling communication between devices for future work, although such issues have been addressed before. For example, the PEN network supported direct peer-to-peer communication, with a scheduling algorithm that was more efficient than the more traditional (centralised) master-slave scheme [WJ02, p. 21].

Both connecting to push notification services via Tor, and the use of Tor hidden services, inherit the anonymity properties of Tor, which is resistant to local adversaries who are able to control any local network. This means that a local adversary does not learn the endpoints of any connections. In addition, the app server may also be located behind a hidden service, providing anonymity for the app server too.

Regardless of whether we use a single push notification server, a push notification service per app, or phone-to-phone communication, our primary concern is that using Tor, and possibly running a Tor hidden service, may be significantly less energy-efficient, or may result in substantially more data usage, than traditional push notification services. Quantifying and improving the cost of Tor is a requirement in all three use-cases and is thus the focus of the remainder of this chapter.

We note that the use of Tor to support push notifications may increase latency for message delivery, but we do not believe the typical latency times found on Tor will lead to large problems for push notifications. We therefore leave this analysis for future work.

4.3 Cost of running a hidden service

We present a series of experiments to measure data usage requirements and to estimate the energy consumption of using Tor and operating a Tor hidden service on smartphones. Our testbed consists of two Nexus 5X smartphones running Cyanogenmod (Android 6.0.1). To support the creation and operation of Tor hidden services, we developed a simple custom app that uses Tor project's Orbot Android app (version 15.1.2) to run a hidden service. Our app accepts connections to the hidden service and logs any data sent to it, allowing us to explore data transmission at various rates between the smartphone and another computer. To avoid problems with the phone going into deep sleep, we configured the phone to always stay awake. To provide a comparison with Google's Cloud Messaging (GCM) service, we installed and enabled the Google Play Services Framework when necessary.

We obtained full packet traces of all traffic on a Linux workstation by connecting the smartphones to a NETGEAR WiFi access point with an Ethernet uplink connected to a workstation. The workstation was configured to route data onto the wider Internet, allowing connections to and from the Tor network and GCM. The experiments where conducted between December 2016 and February 2017.

4.3.1 Measuring Tor traffic

To estimate the cost of using Tor for push notifications, we wanted to construct an empirical model of Tor traffic. Such a model is important for accurately estimating the data and energy costs an app might generate using any of the Tor-based push notification systems we discussed in Section 4.2.

As discussed in Section 2.6.1, the Tor client takes part in many different network activities which we break down into nine categories in order to build an empirical model:

- regular downloads of the network status;
- relay (micro) descriptor data;
- creating circuits to introduction points;
- regular uploads of hidden service descriptors;
- sending keep-alive messages along established connections to Tor relays;
- downloading authority certificates;
- measuring circuit timeouts;

- establishing and closing connections to a (first hop) Tor relay; and
- creating circuits, responding to connection requests, and data communication associated with a hidden service.

In this section we describe how we quantify the amount of network traffic in each above categories. We use this analysis in Section 4.5 to derive an empirical model of Tor data usage and assess the real-world impact of using Tor with handsets in the Device Analyzer project.

Tor traffic is encrypted, and thus it is not straightforward to obtain a breakdown of traffic by category. We therefore instrumented the Tor source code to identify and log the purpose (thus category) of all network data sent or received by the Tor client. We used the log to associate this category with each packet in the network trace captured by the workstation.

Tor clients and routers communicate with one another via TLS connections with ephemeral keys. Traffic on these connections consist of 514-byte *cells*, which contain a header and a payload. Cells are either control cells, used to create, extend, or destroy a *circuit*, or are payload cells, containing encrypted data travelling over an existing circuit. The circuits themselves are used to support connectivity for client applications (e.g. allowing an app on the phone to make a TCP connection to a push notification service) and maintain connectivity to the Tor network (e.g. downloading the network status; uploading a hidden service descriptor; sending a keep-alive message; and so on).

Our instrumented Tor client generally allows us to determine the purpose of each cell sent or received, but associating this with the network trace captured by the workstation is difficult because: multiple cells may be carried inside a single IP packet; a single cell may be split across IP packets; and TLS handshake messages, TLS headers, TCP headers and TCP retransmissions introduce additional overhead that should be associated with the underlying category of use.

Accounting for the TCP header size and retransmissions is relatively easy as these are visible in the packet trace. To account for TLS headers and overheads, we record the number of bytes read and written to the TLS stream, and to the underlying TCP socket. We match the byte counts written to the TCP socket with the bytes sent in the network trace to determine which cells (or parts of cells) are contained within a specific network trace. The overheads resulting from TLS and TCP are assigned proportionally to the cells contained within the relevant packets.

Determining the purpose of each cell is straightforward in most cases since the cell header associates the cell with a specific circuit, and additional instrumentation allows us to record the current purposes of a circuit or of the stream associated with the cell. One complication is that the assignment of a purpose to a cell cannot be made directly

after data is read from the underlying TLS connection, since only part of a cell may be returned. Additional bookkeeping is thus needed so that the purpose can be determined after complete cells have been received and parsed. Another difficulty is that many TCP streams can be multiplexed down a single circuit. For circuits that were used for more than one purpose, some traffic can exist that cannot be assigned to a particular TCP stream (e.g. creating a new circuit); if the purpose cannot be uniquely inferred, the traffic cost is shared equally between all the purposes associated with the circuit.

Consequently, there are two approximations in our analysis that are small and therefore do not have a material impact on our analysis. First, since Tor preemptively builds circuits, some of these circuits may not have been used; we find unused circuits were responsible for only 0.1% of the total traffic. Second, when cells cannot be associated with a TCP stream, and their purpose cannot be inferred, we assign their cost equally to all purposes associated with a given circuit; this only affected 0.2% of the total traffic. Section 4.4.1 offers more details.

4.4 Results

We now report on four experiments. First we measure the cost of maintaining a Tor hidden service for a fixed IP address and stable Internet connection. We then measure the additional cost of changing our IP address, a regular occurrence for a smartphone as it moves between cellular data and WiFi networks. Third, we explore the overhead of data transmission across the Tor network. These results allow us to produce a model of the cost of running a Tor hidden service, something we build on in Section 4.5. Finally, for comparison, we measure the overheads of using GCM.

4.4.1 Hidden service maintenance

We measured the network traffic induced by maintaining a Tor hidden service over a 48-hour period using our testbed. We recorded 32.5 MiB of Tor traffic, including IP headers across 46,790 packets, or an average of 693 KiB (975 packets) per hour. The large majority of traffic volume in bytes was caused by network status consensus downloads (79.9%), with another 11.7% caused by hidden service descriptor uploads. Downloading relay descriptors caused 4.3% of traffic, keep-alive messages 2.5%, and introduction circuits 0.2%. Establishing and closing connections to entry (first hop) relays was responsible for 0.8% of traffic. Measuring circuit timeouts used 0.2%, another 0.2% were used to fetch authority certificates, and the remaining 0.1% were used to manage circuits that remained unused. Table 4.1 provides further detail.

At the time of writing, directory authorities vote on a new network status consensus every hour, which is valid for three hours. Clients download a new consensus at a randomly

Type of traffic	KiB/h	KiB%	Packets/h	Packets%
Network status download	554	79.9%	694	71.2%
Relay descriptors	30	4.3%	47	4.9%
HS descriptor	82	11.7%	144	14.8%
Keep-alive	17	2.5%	54	5.6%
Introduction circuits	1	0.2%	3	0.3%
First-hop connections	6	0.8%	24	2.5%
Measure circuit timeout	2	0.2%	3	0.3%
Authority certificate	2	0.2%	3	0.3%
Unused circuits	1	0.1%	1	0.1%
Total	693	100%	975	100%

Table 4.1: Average network traffic generated when maintaining a Tor hidden service.

chosen time between 105 and 170.6 minutes after their current consensus becomes valid. We observed a total of 38 consensus downloads, with an average size of 699 ± 9 KiB. In addition, we saw one case where the directory server returned a "304 Not modified" status. In this case, the client retried the download after one minute, resulting in the same status code. When the client tried again at a different server 10 minutes later, it received a full consensus document again. This caused an additional 8 KiB of traffic. We also observed 336 hidden service descriptor uploads. Keep-alive messages are padded to the size of a cell, with the total size of keep-alive IP packets being 595 bytes, which is answered by an ACK packet of 52 bytes. Both sides of the connection send a keep-alive packet, resulting in 4 packets and 1 294 bytes exchanged per idle connection every 5 minutes.

4.4.2 Network connectivity changes

Smartphones regularly change their network connectivity as they move between WiFi access points and connections via cellular data services. Whenever such device connectivity changes, connections to the Tor network must be re-established because the source IP address used to support the TCP connections underlying Tor circuits changes.

To estimate the total additional network traffic caused by network connectivity changes, we ran the same measurements as in Section 4.4.1, while periodically forcing network connectivity changes, and compared the results. We forced a disconnect of the WiFi connection every 20 minutes, and a reconnect 5 seconds later. When Orbot detects that the network is down, Tor shuts down all connections and starts rebuilding connections when connectivity is back. We then measured the amount of traffic generated over 48 hours and classified it as in Section 4.4.1. Our experiments showed that network status document and relay descriptor downloads were not affected by connectivity changes. We therefore exclude traffic classified as one of these categories from the comparison to reduce noise. The version of Orbot we used chose new introduction points after each reconnect,

Interval	1 B	512 B	1 KiB
1 min	2.7(7.6)	3.2(7.8)	3.8(9.2)
8 min	5.7(15.7)	6.1(16.4)	7.2(19.6)
12 min	9.1(25.1)	9.9(26.7)	9.7(25.4)

Table 4.2: The average additional network traffic in KiB (number of packets) generated per message over Tor for different message sizes and different sending rates.

and re-uploaded the hidden service descriptors. Based on Section 4.4.1, which describes the traffic required for a set of hidden service descriptor uploads, we also exclude traffic related to them to get an estimate of the remaining traffic caused by a connectivity change. Ignoring traffic related to these three activities, we calculated the difference in total traffic compared to the idle connection (Section 4.4.1). Excluding these, we measured 5 628 KiB of traffic, compared to 1 362 KiB for the idle connection. During the 48 hour period, the WiFi reconnected 143 times. We therefore estimate an average additional traffic per reconnect of 29.8 KiB, primarily for re-establishing connections, introduction circuits, and other circuits. Adding the approximately 70 KiB it takes to upload hidden service descriptors, a reconnect generates roughly 100 KiB of traffic.

4.4.3 Data transmission

We measured the overhead of transmitting data over the Tor network. To do so, we sent messages of three different sizes (1 B, 512 B, 1 KiB) at three different intervals (1 min, 8 min, 12 min) to one of our smartphones. We chose 8 and 12 minute intervals to explore the effect of circuit rebuilds, which currently occur every 10 minutes (Section 2.6.1). For each message, we established a fresh TCP connection to the hidden service and sent a stream of bytes of the given length before closing the connection. For each combination, we sent messages for 4 hours. Table 4.2 shows how much traffic was generated on average by a single message for different message sizes and rates. We estimated this amount by counting all traffic not labeled as network status download, relay descriptor download, hidden service descriptor upload, certificate authority download, or measuring circuit timeout over the 4 hour-period, subtracting the expected amount of traffic for the same categories for maintaining the hidden service as measured in Section 4.4.1 (429 bytes in 1.4 packets per minute), and dividing by the number of messages sent. Note that we count keep-alive traffic, as receiving messages may reduce or increase the need for keep-alive messages.

4.4.4 Comparison with GCM

For comparison, we used our testbed to measure the costs of maintenance, connectivity changes, and message overhead of using GCM. To determine the traffic relevant to GCM,

we filtered TCP traffic from the smartphone whose destination was mtalk.google.com, ports 5228–5230.

Push notifications over GCM requires Google Play Services (PS) running on the handset. PS initiates and maintains a single open TCP connection to a GCM server to receive push notifications. To keep the connection alive, PS periodically sends keep-alive messages to a GCM server. The active keep-alive intervals can be determined by typing the code `*#*#426#*#*` in the Phone app. Using this technique, we experimentally confirmed that, for mobile data connections, PS used a 28-minute interval. On WiFi, PS uses a proprietary adaptive algorithm to determine an interval of between 110 seconds and 29 minutes; in our case the interval was typically set to 19, 24, or 29 minutes.

There were no entries in the smartphone system log concerning keep-alive messages. Thus, to quantify data usage and packet count for keep-alive messages, we looked at the packet trace from the smartphone deployed with our testbed with PS installed and enabled. To ensure that PS connected to GCM and waited for push notifications, we wrote and launched a simple app that waits for incoming GCM messages. We observed a periodic burst of three or four packets with a total length between 224 and 278 bytes, which matched the WiFi heartbeat interval. From the 246 bursts we observed, the average total size was 238 ± 22 bytes (not counting duplicate packets). Alongside this periodic burst, we sometimes observed up to four additional packets containing duplicate TCP packets (up to 528 bytes in total). The contents of the packets were encrypted, so we could not determine further details of the keep-alive message or the purpose of the retransmission. The average total size including duplicate packets was 258 ± 57 bytes.

We repeated the experiment described in Section 4.4.2 for GCM on a Nexus 5X handset. We again forced the phone to reconnect to WiFi every 20 minutes. We ran the experiment for 48 hours. We measured the amount of traffic within a minute after each reconnect and observed a burst of traffic, with an average size of 2.9 ± 1.5 KiB (16.8 ± 1.6 packets) in 141 out of 143 cases when the phone reconnected to WiFi. In two cases, we observed no additional traffic directly after a reconnect. We assume that the fact that we did not change the IP address might have resulted in PS not reconnecting in these cases.

To measure the traffic overhead when sending messages to the smartphone, we sent similar messages to our GCM-enabled app as we did over Tor in Section 4.4.3. We used the same message sizes and intervals (1 min, 8 min, 12 min; 1 B, 512 B and 1 KiB) and we measured each combination for 2 hours. The average traffic per message did not noticeably differ for different intervals. Per 1-byte message we observed on average 0.3 KiB, per 512-byte message, 0.8 KiB, and per 1024-byte message, 1.3 KiB of traffic.

4.5 Empirical model

In this section, we use the results of Section 4.4 to derive a model for the data usage of a hidden service on a smartphone. We use this model to evaluate the data usage and energy costs of using Tor to support a push notification service on real devices in Section 4.5.1. The model is based on the results from our measurements and therefore on the state of the Tor network at the time of our experiments. Future work could take into account the changing nature of the Tor network and create a model that depends on parameters like the number of Tor relays that notably impact the amount of network traffic.

To estimate the total network traffic required to maintain the hidden service on a phone, we require knowledge of the connectivity profile of a device: the periods the device is connected to the Internet via WiFi or a cellular network, when the IP address of the handset changes, and when no network connectivity is available. Network traffic is generated by periodic network status and relay descriptor downloads, hidden service descriptor uploads, and the creation and maintenance of Tor circuits to introduction points. In the following, we look at each of these in turn.

The network status document is downloaded at regular intervals. Building on our analysis in Section 4.4.1, we assume that the Tor client starts a network status download when either: a disconnected device connects to the Internet and has no valid network status document; or time t (chosen uniformly at random from the interval $[105, 170.6]$ minutes) has passed since their current download became valid. We assume that a new network status document becomes valid on the hour, every hour (in UTC) and the client always downloads the most recent valid document. We assume that each consensus download produces 716 419 bytes of traffic, the average measured in Section 4.4.1.

We assume that the client downloads a set of relay descriptors immediately after it downloads a network status document. We assume that this requires $30\,356 \cdot h$ bytes of traffic, where h is the number of hours that have passed since the last network status download. This is based on the average descriptor download traffic we observed per hour, and should give a good approximation, especially because the large majority of descriptor downloads happen shortly after a network status download.

We assume that each time the phone changes the way it connects to the Internet, it needs to rebuild its Tor connections and circuits (including those to the introduction points), which costs 30 548 bytes of traffic, the average measured in Section 4.4.2.

We assume that the client uploads its hidden service descriptor immediately after it has (re-)established its connections to the introduction points, or 60 minutes after the last upload. We assume that each set of uploads (to six directories) incurs 71 504 bytes of traffic, the average measured in Section 4.4.1.

Finally, we assume that for every 5 minutes the device is connected to the Internet, 1 474 bytes of keep-alive traffic is generated, the average measured in Section 4.4.1. We do not include periodic changes of the introduction points, as these have a small impact on total traffic (0.2% during the experiment described in Section 4.4.1) and in our analysis, the Tor client changes introduction points once a day. Similarly, for simplicity, we do not include other traffic in our model since the remaining traffic was only approximately 1% of the total traffic during our measurement.

4.5.1 Evaluation

To evaluate the energy and data usage costs of using Tor to support push notifications, we use our model from Section 4.5 together with connectivity profile data of smartphones from the Device Analyzer project [WRB14a].

Device Analyzer is an Android app, available on the Google Play Store since May 2011, and has been installed on over 30 000 handsets. It gathers a wide variety of system statistics, including: app usage; metadata on calls placed and received; metadata on text messages sent and received; Bluetooth devices seen and connected to; WiFi access points seen and connected to; cell network coverage for calls and data; and battery and power usage. Data collected by the app is processed on the handset to obscure direct personal identifiers (e.g. phone numbers) before uploading data to a server at the University of Cambridge.

We analyzed traces from the 30 444 devices in the Device Analyzer dataset. We excluded devices with less than 30 days' worth of data. We further excluded devices where Device Analyzer data collection had been interrupted at any point, had large jumps in their device clock, or where the device clock was obviously wrong or broken. For each device trace from the remaining 2 014 devices, we estimate the volume of cellular data required to maintain a Tor hidden service. We do this by assuming that cellular data is used when a cellular connection is available and a WiFi connection is not. Since the Tor client uses timing randomisation when downloading the network status document, we simulate the connectivity pattern of the device 40 times and take the average amount of traffic.

The baseline violin plot in Figure 4.1 shows our estimate of the cost of running a Tor hidden service for 30 days on the 2 014 devices from the Device Analyzer dataset. An equivalent numeric summary is shown in Table 4.3. Cellular data usage is high, with a median cost across all devices of 198 MiB. For 10% of the devices we estimate a cellular data usage of 362 MiB or more. By way of comparison, GCM maintenance, without any IP address changes, costs on average 258 bytes every time it needs to send a heartbeat (Section 4.4.4), or 0.44 MiB over 30 days for heartbeat interval of 24 minutes. Even

factoring in multiple network changes per day, at 2.9 KiB per change, total costs would still likely be only a couple of MiB per month.

We use EnergyBox [VNTP14] to estimate the energy costs of maintaining a Tor hidden service on a smartphone. EnergyBox takes a packet trace, a smartphone model, and a connectivity profile (WiFi or cellular). To provide a reasonable lower-bound estimate for the energy costs, we reused the 48-hour packet trace collected for our experiment in Section 4.4.1 and assumed this trace was transferred over the cellular data network (3G) with a Nexus One device on the TeliaSonera network, the only device and network operator the EnergyBox authors provide an energy model for. The Nexus One was released in 2010, and we expect newer devices' batteries to last longer. Over the 48-hour period, the estimated total energy costs were 5 346 J; or 2 673 J = 0.743 Wh per day. The Nexus One device has a battery capacity of 5.18 Wh, so, assuming a battery profile where the device is charging for 8 hours over night and on battery for 16 hours, this represents 9.6% of total battery capacity – a significant amount. Note that this value only takes into account the energy required for network communication, and additional power is required to make a hidden service work, e.g. to keep the device awake when needed.

4.5.2 Reducing Tor data usage

The above numbers demonstrate that running a Tor hidden service on a smartphone generates several hundreds of megabytes of cellular data traffic per month on a typical device – an unacceptable volume for all but those with a generous data package. As the EnergyBox paper [VNTP14] demonstrates, there is a strong correlation between data volume and energy usage. Therefore, we evaluate four strategies to reduce the amount of data that Tor requires, thereby reducing energy usage. These strategies may impact user anonymity. We leave evaluating this for future work.

Strategy A: Reconnect to the same introduction points

In the version of the Tor client we used in our experiments, Tor chose a new set of introduction points whenever a device changed its IP address. This is a known issue that is supposed to be fixed [torg]. However, we found that changes in network connections continued to require new introduction points. As of December 2018, a fix was under development [torf].

Changing introduction points requires the Tor client to generate a new hidden service descriptor and upload it. This causes additional traffic, and also affects anyone wanting to connect to the hidden service: other clients may have cached the hidden service descriptor and therefore connectivity is broken. At the time of writing, this was still an open issue in the Tor bug tracking database [tore].

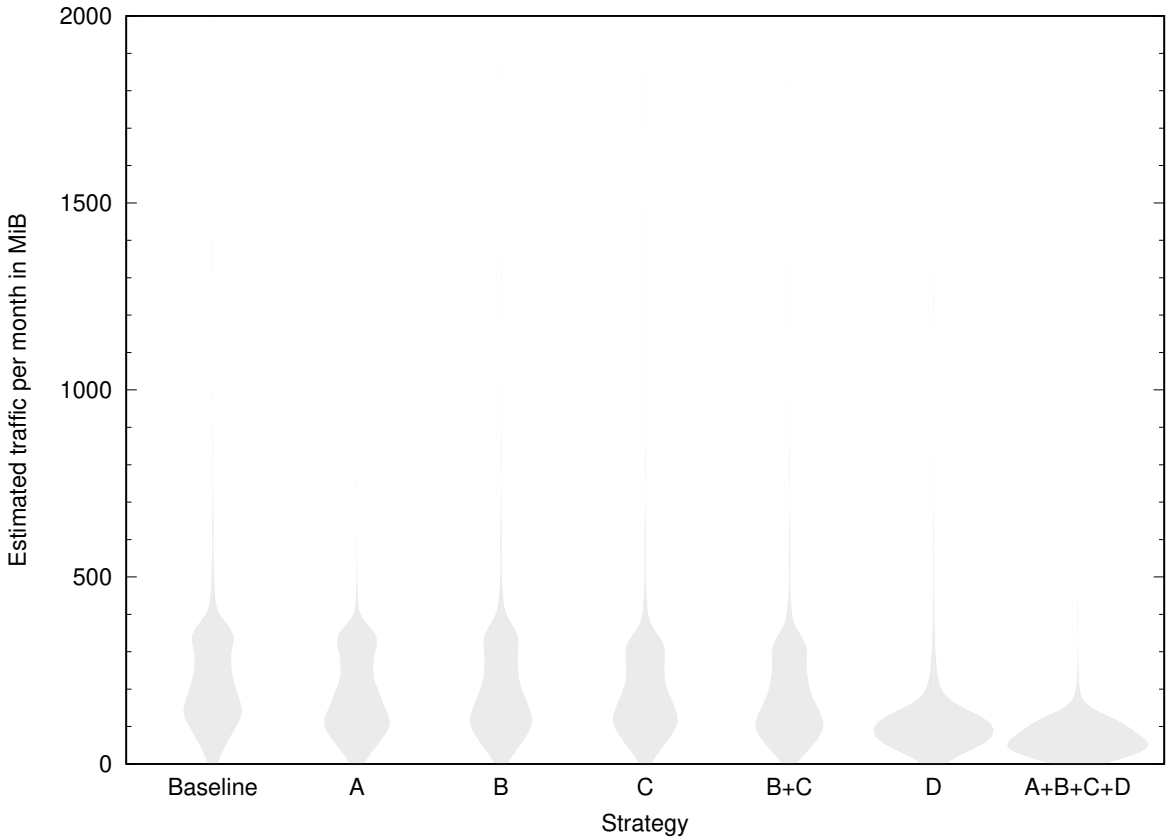


Figure 4.1: Estimated traffic over the cellular network over 30 days for 2 014 devices in the Device Analyzer dataset. The leftmost violin plot shows a data usage estimate for Orbot's current behaviour; the remaining ones estimate data usage with various data reduction strategies developed in Section 4.5.2.

Strategy A in Figure 4.1 estimates the data usage costs for a Tor client that reconnects to the same introduction points when the IP address of the smartphone changes, and therefore does not need to re-upload the hidden service descriptor.

Strategy B: Proactively fetch network status on WiFi

If we assume free data usage over WiFi, a straightforward strategy to reduce cellular data costs is to proactively download the most recent network status document as soon as it is available. This has the downside of causing additional traffic at the directory mirrors and will also increase energy costs. Strategy B in Figure 4.1 explores this option.

Strategy C: Defer fetching network status on cellular

Since mobile devices regularly move between WiFi and cellular data, it makes sense to delay downloading the network status document until just before expiry in the hope that WiFi connectivity appears before a download over the cellular network becomes necessary. This may additionally reduce the total number of network status document downloads.

	Base	A	B	C	BC	D	ABCD
Median	198	165	172	169	154	95	61
Mean	367	305	333	322	304	188	119
Std.dev.	141	108	138	132	132	101	45
90^{th} perc.	362	341	353	329	327	172	120
99^{th} perc.	649	424	628	612	607	517	202

Table 4.3: Estimated traffic in MiB over the cellular network over 30 days for 2 014 devices in the Device Analyzer dataset.

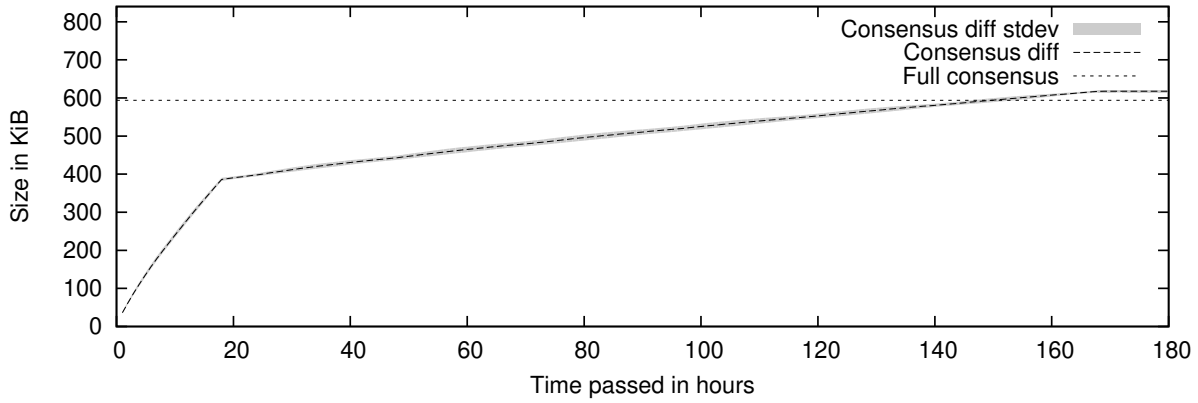


Figure 4.2: Average size of the compressed output of `diff -d -e` on pairs of all microdescriptor consensus documents from 1 November to 30 November 2016, compared to the compressed size of the full network status documents.

This strategy may cause spikes in download requests on directory mirrors in the Tor network if many clients adopt the policy. Strategy C in Figure 4.1 explores this option.

Strategy D: Download network status diffs

The network status document is updated every hour, but only a small part of it changes from hour to hour. Therefore, we consider the potential benefits of downloading the difference between two network status documents. This is not a new idea [Pal08] – a version of this was implemented in 2014 [Mar14] – but it had not yet been integrated into the main branch of Tor or available in Orbot at the time of our experiments. To estimate potential savings from downloading differences instead of full documents, we looked at all 720 consecutive network status documents from November 2016 (UTC).

We compared documents pairwise by applying `diff -d -e` and then `gzip -9` to the output to calculate the size of a diff. We calculated how the size of the diff changed as the time period between the pairs of documents increased. Figure 4.2 shows the average size of the compressed difference between all pairs of documents, grouped by the number of hours between the start of the validity of the two documents. The knee in the curve at 18 hours is due to the fact that Tor relays update their relay descriptors every 18 hours. The data shows that using diffs can drastically reduce download size if time between network status

downloads is small. The diff between two consecutive network status documents is a mere 6% of the size of the full network status document; it is still 35% smaller after 18 hours, reaching the size of a full network status download only after 6 days. Thus, consensus diffs can be particularly beneficial for devices that constantly stay connected to the Tor network and frequently download the consensus. To account for the overhead incurred by downloading the diffs, we add 17.6% to the sizes of all compressed diffs for our model. We calculated this percentage by comparing the average traffic incurred for a consensus download in Section 4.4.1 with the average size of compressed consensus documents from November 2016.

Strategy D in Figure 4.1 computes the cost savings on cellular data if network status diffs follow the averages found in Figure 4.2.

4.6 Related work

Previous work has highlighted a scalability problem in Tor's design: every client needs up-to-date information on all relays, resulting in a total bandwidth expenditure that grows with the number of clients times the number of relays [MTHK09, MOT+11]. Several papers propose more scalable solutions using peer-to-peer architectures [MTHK09, NW06, PRR09, RP04, MB09]. These approaches are usually based on using distributed hash tables (DHTs) and/or random walks letting clients find random relays on demand without needing the entire list of relays. Mittal et al. [MOT+11] proposed an alternative approach to improve Tor's scalability: keep the existing client-server architecture, and let clients obtain random relays from directory servers or guard relays using private information retrieval techniques. While the focus was to reduce the data usage from network status downloads on the Tor network centrally, such techniques also offer benefits to mobile devices with limited data usage requirements and energy constraints.

Loesing et al. [LSWW08] measured the time taken to complete the steps involved in connecting to a hidden service. Lenhard et al. [LLW09] conducted similar experiments with similar results. They also measured the time a Tor client takes to complete the bootstrapping phase under low-bandwidth conditions. Solberg and Bezem [SB13] measured various performance characteristics of the Tor network and Tor hidden services, including throughput, access time, connection latency, and reliability, for both the public and a private Tor network.

Wiangsripanawan et al. [WSSN07] and Doswell et al. [DAKS13, DKAS15] looked at the impact of mobility on the performance of Tor in client mode. They explored the problem of changing network connectivity, and the resulting change in IP address, the need to re-establish Tor circuits, and the loss in connectivity. Wiangsripanawan et al. also considered location privacy. To keep connections alive across changes of IP address, they

propose re-establishing a circuit to the same exit node. Doswell et al. estimated the impact on throughput, and proposed client throttling to reduce the amount of "wasted" traffic, and the use of a trusted (private or public) bridge relay to keep circuits open, allowing the client to quickly reconnect to the circuits. Neither of these papers consider the costs of running a hidden service from a mobile device.

Briar[2] is an open source app that uses Tor hidden services to support instant messaging between smartphones without using other cloud infrastructure or push notification servers. They do not quantify the costs of running a hidden service; the results from our work provide support for the introduction of network status diffs.

4.7 Discussion

A few months after we published the results presented in this chapter [KB17c], Tor developers finished implementing consensus diffs; the changes have been merged into the main development branch in May 2017 [tord]. Based on the results presented above, we therefore expect that data usage of maintaining a Tor hidden service has been reduced with more recent versions of Tor. Since the number of Tor relays and the size of the consensus document have remained nearly the same since our experiments, we expect that the median cellular data usage has now roughly halved to around 100 MiB, assuming the connectivity patterns of typical users have also remained similar.

However, there remains significant work to be done. Our experiments show that Google Cloud Messaging costs in the order of 1 MiB per month, nearly two orders of magnitude less than a Tor hidden service with all four of our data reduction strategies deployed. Similarly, transmission of a single 1 KiB message consumes significantly more over Tor (between 3.8 KiB and 9.7 KiB of data; see Table 4.2) as compared with GCM (1.3 KiB; See Section 4.4.4).

The introduction of Doze [And] in Android 6.0 makes some form of privacy-preserving push notification service all the more important. Doze is enabled when the handset is stationary for a period of time, not charging and the screen is off. When Doze is enabled, a handset conserves battery life by suspending apps, including suspending background tasks, network communication, alarms, and wake locks. Consequently, GCM and its successor, FCM, is an essential developer tool if an app needs to receive messages from an external source because high-priority messages sent over the network to Google Play Services are not affected by Doze.

Some popular apps such as Facebook already allow users to connect via Tor from their smartphone. However, they currently lack support for push notifications for these connections. With some further work to Tor as outlined in Section 4.5.2, using Tor for

[2] `https://briarproject.org/`

push notifications may be acceptable for such users. Tor hidden services are also likely to be useful to support direct phone-to-phone communication in next-generation apps such as Briar.

Scaleability is a concern if Tor hidden services become popular for deploying push-notification services. If millions of mobile devices start running a Tor hidden service, the increased bandwidth requirements on the Tor network and the large number of hidden service descriptors that need to be managed by hidden service directories will require more Tor relays to be deployed. This will in turn increase the size of the network status document that needs to be downloaded to the devices regularly. Future work is therefore required to develop more scalable anonymity networks.

Data and source code used to produce the results in this chapter are available [KB17b]. Data from Device Analyzer is already available from the Device Analyzer project.

4.8 Ethical considerations

Experiments were conducted on the live Tor network and using anonymized personal data from the Device Analyzer project. In the following we shortly discuss how this was done according to ethical principles for research.

The Device Analyzer project has been approved by the Ethics Committee of the Department of Computer Science at the University of Cambridge. It is designed to protect user privacy. For example, all personal identifiers in the dataset are replaced by a salted hash [WRB14b]. This dataset is available to researchers under the condition that they do not attempt to re-identify users.

We used the dataset to obtain a realistic distribution of network connectivity patterns and made no such attempts and did not link the data with any other dataset containing personal identifiers.

We captured network packet traces from the live Tor network. In particular, we captured network traffic between contained the encrypted network traffic between our experiment phone and Tor entry nodes. Additionally, we did not store the cryptographic keys necessary to decrypt any of the traffic, thus the only usable data from the traces is metadata about the packets exchanged with Tor entry nodes. This metadata contains the IP addresses of the Tor entry nodes, as well as timing and size of packets. IP addresses of Tor entry nodes are publicly known and thus not sensitive. The timing and size of packets was needed for our research, and this information is not sensitive either since it only relates to our own traffic and would not give an adversary any significant advantage in de-anonymizing other users.

4.9 Summary

In this chapter, we showed that using Tor, the cellular data cost of maintaining a Tor hidden service from a smartphone today is high, with a median cost across all devices of 198 MiB in the version we used. In the worst case, we expect devices with monthly cellular data usage in excess of 600 MiB. Energy costs were also significant: we estimated the network activity would cost at least 9.6% of battery capacity on a Nexus One connected to the Internet via 3G with a daily charge cycle.

We explored four cost-reducing strategies for maintaining a Tor hidden service on a smartphone: reconnect to the same introduction points when the phone's IP address changes; proactively fetch network status on WiFi; defer fetching network status on cellular connections; and download network status diffs. Combined, these four strategies result in a more reasonable total monthly median cost of 61 MiB. The most effective of the strategies, consensus diffs, has been added to Tor after our results were originally published.

CHAPTER 5

Authenticated snapshots

In this chapter we describe how to build a document collaboration system that supports concurrent edits by several user devices, and that does not require a server to receive and centrally process all edits. Our protocol supports end-to-end encryption of all communication between user devices. Moreover, it allows devices to communicate peer-to-peer via an anonymisation network such as Tor [DMS04] or Loopix [PHE+17], allowing the collaboration activity to be hidden from an attacker who may be performing traffic analysis.

Without a trusted server that authenticates users and holds the authoritative version of the shared document, user devices need another means of verifying the integrity, authenticity, and consistency of edits they receive from other users. For example, consider a scenario where lawyers from different companies are negotiating a contract using a shared document editor. It is essential that everyone has a consistent view of the document and that changes can be traced back to the author. In Section 5.2, we describe the system architecture, including the design goals and threat model. In Section 5.3, we first present a simple protocol that uses digital signatures and cryptographic hashes to verify the origin of an operation, and to efficiently check if all devices have seen the same set of document edits. This protocol is sufficient if the set of participants in a collaboration group is fixed or if new collaborators can receive the full editing history. If it is not desirable that new collaborators can see all changes that happened before they joined and parts that have been deleted, the protocol presented in Section 5.4 can be used, which extends the basic protocol by support for authenticated snapshots. Any device can invite new collaborators by sending them an authenticated snapshot created from the current document state. The new collaborator can verify that the snapshot is consistent with views by other devices, and start collaborating with the group, without being able to see edits that happened before they joined.

We make use of operation-based CRDTs (Section 2.1.2) in order to ensure that concurrent edits to a document can be merged by user devices without conflicts. We have chosen CRDTs because they are decentralised by design and do not need a central server – data can flow directly between devices. This model contrasts with the Operational Transformation approach to document collaboration [NCDL95, EG89], as used in Google Docs [DR10], which requires a central server.

After presenting the protocol in this chapter, in Chapter 6 we evaluate its costs and its correctness and security properties.

5.1 Adding new collaborators

Designing a collaborative editing application becomes more challenging if new collaborators are invited to join the editing session for an existing document. In this case, the new collaborator must be given a copy of the document at the time she is invited, and then be sent any subsequent edits to the document. We identify three approaches to inviting new collaborators:

1. If existing collaborators keep a log of all editing operations that have occurred since the creation of the document, they can send a copy of that log to the new collaborator, who can then reconstruct the current state of the document from the edits. The new collaborator can also use the hashes and signatures on the operations to verify the integrity of the edit log. This approach is used by the protocol we describe in Section 5.3, but it has significant disadvantages. In particular, storing, transmitting, replaying, and checking the integrity of the edit log incurs substantial costs in storage, network bandwidth, and processing time; and the edit log contains all past versions of the document, including any text that has been deleted in the current version.

2. To reduce the cost and improve the privacy properties of the first approach, the new collaborator could be sent only a *snapshot* of the current state of the document, not including its past editing history. To ensure consistency, each existing participant can be asked to confirm the validity of the snapshot. However, if any existing participants are offline, the new participant must either wait (potentially indefinitely) until they are next online, or go ahead and accept the risk that its snapshot is inconsistent with other participants' view of the document.

3. To overcome the downsides of the first two approaches, we develop a new protocol in Section 5.4. In this protocol, new collaborators are only sent a snapshot of the current state of the document, plus a cryptographic proof of the integrity of the

snapshot. The new collaborator can then use this proof to verify the snapshot, without having to wait for any communication with other participants.

This chapter describes a scheme that allows authenticated snapshots to be created by devices collaboratively editing a document in a peer-to-peer setting. The solution allows devices to verify that a snapshot represents a state of a document that is consistent with (possibly earlier) states of the document that have been signed by all collaborators.

5.2 System architecture

We envision a collaborative document editing system as described in Chapter 3, with an arbitrary number of users, each of whom may own one or more devices. Document editing software is installed on each device, allowing the user to create a new document, invite others to collaborate on a document, and join an existing document. The software allows users to edit any document regardless of whether they are currently connected to a network or not; if no network connection is available, then document changes are applied locally and sent to peers when network connectivity returns. Modelling typical mobile devices, we assume that devices may frequently be offline, and that devices may suffer a permanent failure without warning; e.g. if dropped in water.

To protect document editing metadata from traffic analysis, we assume that all communication takes place via an anonymous communication network such as Tor [DMS04] or Loopix [PHE+17]. We assume that an existing key exchange and encryption protocol protects the confidentiality of messages sent via the network. In addition, we assume a public key infrastructure, such as CONIKS [MBB+15], which is able to map human-readable names, such as cellphone numbers and email addresses, to public keys. This infrastructure is used by the software to allow users to invite collaborators by looking up the public key associated with a human-readable name.

5.2.1 Design goals and threat model

We assume the adversary is able to control network communication and can read, modify and delay any traffic, including partitioning the network for arbitrary periods of time; however, the adversary is unable to break the anonymity of the underlying anonymous communication network. Further, we assume the adversary can create an arbitrary number of fake users with devices that may participate in group collaboration; these devices may deviate arbitrarily from the protocol. The adversary cannot compromise the public-key infrastructure and does not have access to secret keys of honest participants; therefore the adversary cannot forge messages or signatures created by honest participants.

We assume that an existing encryption layer in the underlying communication network protects the confidentiality of messages between participants. On top of this network, our protocol provides the following properties in the face of the adversary:

Edit integrity. The shared document can only be modified by a group member.

Attributability. All edits are attributable to the honest device that made the modification. Group members can identify who added a certain part of a document, even if it was added before they joined.

Consistency. Devices have consistent views of the document. When an honest device processes an edit operation, it must have previously processed exactly those edits that happened before this operation, and possibly some concurrent edits.

Snapshot consistency. On joining a group, a new member can check the integrity of the document, i.e. they can verify that the state is consistent with states seen by other collaborators. In particular, they can verify that all modifications made or seen by collaborators up to a certain point are represented in the snapshot, and that no modifications were falsely attributed to a collaborator.

Edit history privacy. A new group member cannot see edits made before she joined the group, other than what can be inferred from the document state when she joins; in particular, she cannot see parts of the document that were deleted before she joined.

Convergence. When honest group members communicate, their local copies of the shared data converge towards a consistent state, even if arbitrarily many group members are malicious.

Availability. Any two participants can collaborate on a document, even if all other collaborators are offline; in particular, the protocol does not require any quorum of devices to be reachable.

Scalability. Assuming a bounded number of collaborators, protocol messages add only a constant communication overhead compared to a simple protocol that does not allow authenticated snapshots; communication and computational overhead for inviting a new member, sending and processing a snapshot is practically linear in the number of atoms in the document at the time of the snapshot.

We prove in Chapter 6 that the protocol described in Section 5.4 satisfies these properties. The properties protect against different kinds of attacks an adversary might attempt. For example, an estate agent selling a house could try to present different views of a contract to different parties, showing different sale prices and keeping the difference.

In a collaborative code editor, an attacker may want to insert malicious code and attribute it to someone else.

Edit history privacy allows, for example, lawyers to collaborate on a contract and later share it with a third party, while ensuring that the third party is unable to see potentially sensitive contents of any previous versions of the contract. Edit history privacy is also useful when researchers working on a paper want to share a draft with a colleague, but would prefer not to reveal previous unpolished versions of the paper.

Our proposed method of collaboration also works if data is shared via a local network, potentially disconnected from the Internet, such as Bluetooth or local WiFi. However, in this case we must assume that the protections afforded by the anonymity protocol are not available.

5.3 Basic protocol

In this section, we describe a basic protocol for collaborative editing of a text document that relies on all collaborators having a copy of the full editing history of the document. In Section 5.4 we will show how to improve the protocol's privacy properties so that new collaborators can be given a snapshot containing only the current document state, and not the past editing history.

The document is initially created on one particular device, and any existing device can add a new collaborating device using an addDevice operation (see Section 5.3.4). We assume that each device has a distinct secret key, and that when a new collaborating device is added, the sender of the invitation can verify the linking of device identifier and public key of the device being added (for example, by means of a PKI). Moreover, we assume that each device is identified by a unique device identifier, *deviceID*, which may for example be a hash of its public key.

5.3.1 Breaking text into atoms

Following the Treedoc algorithm [PMSL09], we represent a collaboratively editable text document by a set of *atoms*. Each atom represents an editable unit of text, for example, a character, a word, a line, or a sentence, and the metadata associated with it. The granularity of atoms can be chosen depending on the application, and does not affect the operation of the protocol.

An atom is a 4-tuple (pos, src, ctr, txt):

- *pos* is a variable-length bit string that identifies a position in the document as in Treedoc (Section 2.1.2.2).

- *src* is the *deviceID* of the source (the device on which the atom was originally created).
- *ctr* is a sequence number that is incremented by the sender as described in Section 5.3.2.
- *txt* is a text fragment (e.g. character, word, or line).

Note that an atom can be uniquely identified both by its position identifier, and by the tuple (src, ctr).

We then define a total ordering on atoms based on the position identifiers:

$$(pos_1, src_1, ctr_1, txt_1) < (pos_2, src_2, ctr_2, txt_2) \iff (pos_1 < pos_2) \tag{5.1}$$

The text of the document is obtained by sorting the set of atoms by this ordering, and concatenating the associated text fragments in that order. We allow the text to be edited through two types of operation: inserting an atom, and deleting an atom. Replacement of text is expressed as deletion and subsequent insertion.

5.3.2 Sending messages

Collaborators communicate by sending and receiving messages. Each collaborator maintains a set of messages it has sent and received; for example, $msgs_A$ is the set of messages sent or received by A.

Each message is a 5-tuple $(src, ctr, op, deps, sig)$, constructed as follows:

- *src* is the *deviceID* of the source (the device that created the message).
- *ctr* is a sequence number that is 1 for the first message sent by a particular *src*, and incremented for each subsequent message from *src*:[1]

$$ctr = \begin{cases} 1 & \text{if } \nexists\, (src, \_, \_, \_, \_) \in msgs_{src} \\ 1 + \max\left\{c \mid (src, c, \_, \_, \_) \in msgs_{src}\right\} & \\ & \text{otherwise} \end{cases} \tag{5.2}$$

- *op* is an operation: either $\mathsf{insert}(pos, text)$ to represent the insertion of a new atom, $\mathsf{delete}(src', ctr')$ to represent the deletion of an existing atom, noop if the document has not been changed, or $\mathsf{addDevice}(deviceID, publicKey)$ to announce the addition of a collaborator device. The noop operation is useful so a device can acknowledge that it has seen a certain state without performing any changes.

[1] We use the underscore as placeholder for a fresh, existentially quantified variable. For example, $(x, \_, \_) \in A$ is shorthand for $\exists\, y, z.\ (x, y, z) \in A$, and $\nexists\, (x, \_, \_) \in A$ is shorthand for $\nexists\, y, z.\ (x, y, z) \in A$.

- *deps* is the set of *dependencies* of this message, that is, a reference to the most recent prior message from each device:

$$\begin{aligned} deps = \big\{ & (s, c, \mathrm{h}(m)) \mid \\ & m \in msgs_{src} \wedge m = (s, c, \_, \_, \_) \wedge \\ & \nexists\, c'.\ ((s, c', \_, \_, \_) \in msgs_{src} \wedge c < c') \big\} \end{aligned} \tag{5.3}$$

  deps is a set of triples consisting of the source *deviceID*, the sequence number of the most recent message seen from that source, and the hash of that message. The hash $\mathrm{h}(m)$ of message $m = (src, ctr, op, deps, \_)$, is computed as a cryptographic hash of the message contents (excluding the signature), and is used to check that all collaborators have received the same message contents:

$$\begin{aligned} \mathrm{h}(m) &= \mathrm{H}(src \parallel ctr \parallel op \parallel deps), \\ \text{where} \quad m &= (src, ctr, op, deps, \_). \end{aligned} \tag{5.4}$$

  $\mathrm{H}(\cdots)$ can be any secure hash function, such as SHA-256. Note that this creates a directed acyclic graph of hashes, where each message references the previous message from the same device and any messages received from other devices. This hash-DAG is similar to the commit history in the Git version control system.

- *sig* is a digital signature of the preceding elements of the message tuple, using the private key of the sender *src*:

$$sig = \mathrm{sign}_{src}(docID \parallel src \parallel ctr \parallel op \parallel deps), \tag{5.5}$$

  where *docID* is a document identifier that uniquely identifies the document. We assume that the document identifier is known to all participants, e.g. through the messaging protocol.

When the source device *src* sends a message m, it adds the message to its message set:

$$\begin{aligned} msgs'_{src} &= msgs_{src} \cup \{m\}, \\ \text{where} \quad m &= (src, ctr, op, deps, sig). \end{aligned} \tag{5.6}$$

m is sent to the other collaborators using a secure messaging protocol, which we elide in this description. Any protocol that protects the confidentiality and integrity of the message against network attackers can be used.

5.3.3 Receiving messages

When a message $m = (src, ctr, op, deps, sig)$ is received by a destination device dst, the destination device performs the following checks:

1. There is no existing message from the same src with a counter value greater than or equal to the incoming message:

$$\forall c.\ (src, c, \_, \_, \_) \in msgs_{dst} \implies c < ctr. \tag{5.7}$$

2. The dependencies are satisfied:

$$deps \subseteq \big\{(s, c, \mathrm{h}(m')) \mid m' \in msgs_{dst} \wedge m' = (s, c, \_, \_, \_)\big\} \tag{5.8}$$

   If $msgs_{dst}$ does not contain the dependencies because they have not yet been delivered, the message m can be buffered locally, and the destination device can request retransmission of the missing messages. Then the delivery of m can be retried after other messages have arrived. However, if the check fails because the hashes are mismatched, m must be rejected.

3. sig is a valid signature of $docID \parallel src \parallel ctr \parallel op \parallel deps$, checked with src's public key.

If all of these checks succeed, m is added to the destination device's message set:

$$msgs'_{dst} = msgs_{dst} \cup \{m\}. \tag{5.9}$$

Assuming second preimage resistance of the hash function and unforgeability of the signatures, the destination device knows that $msgs_{dst} \supseteq msgs_{src}$ if the above checks succeed, since the hashes in $deps$ transitively include all messages in $msgs_{src}$ at the time the message was sent.

Finally, on any device A, the set of atoms $S(msgs_A)$ that make up the document is the set of atoms that have been inserted but not deleted:

$$S(msgs) = \big\{(pos, src, ctr, txt) \mid (src, ctr, \mathsf{insert}(pos, txt), \_, \_) \in msgs \wedge \nexists\, (\_, \_, \mathsf{delete}(src, ctr), \_, \_) \in msgs\big\} \tag{5.10}$$

The text of the document is obtained by sorting this set of atoms as described in Section 5.3.1.

5.3.4 Adding a new collaborator

When an existing collaborator wants to add a new device as a collaborator, it first broadcasts a message containing an addDevice($deviceID, publicKey$) operation to announce to other devices that a certain device has been added. Moreover, the device A that invites the new collaborator must send the entire set $msgs_A$ to the new device. The new device can then check the integrity of these messages by performing the same checks as in Section 5.3.3.

If A is malicious, it may try to make the new device's view of the document diverge from the rest of the group. However, A is limited to two attacks: it can give the new device an old version of the document (corresponding to a subset of $msgs_A$), and it can give the new device a document containing edits that have not yet been sent to other collaborators. In either case, when the new collaborator communicates with other group members, they will exchange the missing operations. If their views of the edit history are inconsistent, they will detect this situation and can then run a suitable resolution protocol to reach consistent states.

5.4 Privacy-enhanced protocol

The protocol described in Section 5.3 has the problem that the full editing history, including any deleted past content of the document, is exposed to a new collaborator when she joins. In this section we present a revised protocol that avoids this problem.

Specifically, we want to be able to send a new collaborator only the current set of atoms, rather than the full set of operations that led to this set of atoms. However, simply changing the protocol of Section 5.3.4 to send $S(msgs_A)$ instead of $msgs_A$ removes any ability for the new collaborator to check the integrity of the document, since it cannot use the checks in Section 5.3.3. Thus, a malicious device could send the new collaborator an arbitrarily corrupted set of atoms.

In order to allow any new collaborator to check the integrity of the set of atoms, we use RSA accumulators [BP97, BD93]. Each device i generates an RSA key pair with primes (p_i, q_i), and makes the modulus $N_i = p_i q_i$ public. Moreover, device i also chooses an x_i with $1 < x_i < N_i$ and makes it public. For practical purposes, x_i can be fixed.

Since the following protocol description contains a considerable number of variables, for reference, Table 5.1 contains a list of the variables used, with a short description and the section where the variable is defined.

Variable	Description	Section
docID	Unique identifier of the shared document	5.3.2
src	Unique *deviceID* of the device that created a message	5.3.2
ctr	Per-device sequence number of a message	5.3.2
op	Edit operation	5.3.2
pos	Position identifier	5.3.1
txt	Atomic text fragment	5.3.1
deps	List of dependencies of a message	5.3.2
sig	Cryptographic signature of a message signed by the originator	5.3.2
$\mathrm{h}(m)$	Hash of message m	5.3.2
r	Per-message random nonce	5.4.1
acc	Accumulator value corresponding to the current set of atoms	5.4.1
T_{src}	Merkle tree of messages from device *src*	5.4.1
$T_{src}[c]$	Merkle tree of messages from device *src* up to $ctr = c$	5.4.1
mr	Set of root hashes of Merkle trees T_{src} for all devices *src*	5.4.1
mh	Hash of *mr*	5.4.1
$msgs_d$	Set of messages sent or received by device d	5.3.2
S_d	Set of atoms that are part of the document view of device d	5.4.1
sdesc	Set of state descriptors for all collaborating devices	5.4.3
mproofs	Merkle consistency proofs proving that devices' views are consistent	5.4.3
mnodes	Subset of Merkle tree nodes required to continue appending leaves	5.4.3
wit	Witness proving which atoms were present in a device's view	5.4.3

Table 5.1: Variables used in the description of the privacy-enhanced protocol, with reference to the section where the variable is defined.

5.4.1 Sending messages

We update the definition of a message in Section 5.3.2 by adding three additional elements: a nonce r, an accumulator *acc*, and a hash *mh*. In our revised definition, a message is an 8-tuple $(src, ctr, op, deps, r, acc, mh, sig)$:

- *src*, *ctr*, *op*, and *deps* are defined as in Section 5.3.2.
- r is a 128-bit random prime.
- *acc* is the value of an RSA accumulator over the current set of atoms $S_{src} = S(msgs_{src})$, which is derived from $msgs_{src}$ as shown in (5.10), and r:

$$acc(S_{src}, r) = x_{src}^{P(S_{src})r} \mod N_{src}, \tag{5.11}$$

$$\text{where} \quad P(S) = \prod_{a \in S} \text{prime}(a). \tag{5.12}$$

The function prime(a) is a hash function that returns only prime numbers, as described in Section 2.3. We accumulate r in addition to the set of atoms to make

the accumulator indistinguishable, i.e. to prevent guessing the accumulated set based on the accumulator value [DHS15].

- *mh* is the hash of a set of Merkle trees, defined as follows. Let T_s be a Merkle tree [Mer88] containing all message hashes received from device s in order of their sequence number (including the current message if s is the sending device *src*), and let $\mathrm{MTH}(T_s)$ be the Merkle tree root hash of T_s. Then

$$mh = \mathrm{H}\Big(\big\{\mathrm{MTH}(T_s) \mid s \text{ is a } \mathit{deviceID}\big\}\Big). \tag{5.13}$$

  In Section 5.4.3 we use this construction to prove that the sequence of messages from a particular sending device is an append-only sequence, following the approach of Certificate Transparency [DGHS16, LLK13]. To ensure that *mh* is unique, the elements of the set are hashed in a fixed order, e.g. in lexicographic order of *deviceID*s.

- *sig* is extended to also cover the accumulator and the Merkle tree root hash. Moreover, instead of *op*, we include $\mathrm{h}(m)$ in the data to be signed:

$$sig = \mathrm{sign}_{src}(\mathit{docID} \parallel \mathit{src} \parallel \mathit{ctr} \parallel \mathrm{h}(m) \parallel \mathit{deps} \parallel \mathit{acc} \parallel \mathit{mh}). \tag{5.14}$$

  This construction allows a new collaborator to verify the signature of a partial message without necessarily knowing the operation contained in the message. The hash of the message, $\mathrm{h}(m)$, is extended by the nonce and accumulator:

$$\begin{aligned} \mathrm{h}(m) &= \mathrm{H}(\mathit{src} \parallel \mathit{ctr} \parallel \mathit{op} \parallel \mathit{deps} \parallel r \parallel \mathit{acc}), \\ \text{where} \quad m &= (\mathit{src}, \mathit{ctr}, \mathit{op}, \mathit{deps}, r, \mathit{acc}, \_). \end{aligned} \tag{5.15}$$

In practice, the accumulator can be maintained incrementally, so it does not need to be recalculated from scratch for every message sent by *src*. When a new atom a is added to S_{src}, the new accumulator can be computed with two modular exponentiations, provided that the sender remembers the accumulator excluding the nonce, $\mathit{acc}(S, 1)$:

$$\begin{aligned} \mathit{acc}(S \cup \{a\}, r) &= x_{src}^{P(S \cup \{a\}) \cdot r} \mod N \\ &= \mathit{acc}(S, 1)^{\mathrm{prime}(a) \cdot r} \mod N \end{aligned} \tag{5.16}$$

When an atom a is removed from S_{src}, we can update the accumulator by computing the multiplicative inverse of $\mathrm{prime}(a)$ modulo $\varphi(N)$. That is, we use the extended Euclidean algorithm to find t and u such that

$$t \cdot \mathrm{prime}(a) - 1 = u\,\varphi(N) = u(p-1)(q-1), \tag{5.17}$$

which can be done efficiently by the device that knows the secret key (p, q). Then, by Euler's theorem, we have

$$x^{t \cdot \text{prime}(a)} = x \cdot \left(x^{\varphi(N)}\right)^u = x \cdot 1^u = x \mod N, \tag{5.18}$$

and hence we can update the accumulator as follows:

$$\begin{aligned} acc(S - \{a\}, 1) &= \left(x_{src}^{P(S-\{a\}) \cdot \text{prime}(a)}\right)^t \mod N \\ &= acc(S, 1)^t \mod N. \end{aligned} \tag{5.19}$$

5.4.2 Receiving messages

When a message $m = (src, ctr, op, deps, r, acc, mh, sig)$ is received by a device dst, it first performs the same checks as in Section 5.3.3.

Next, to validate the accumulator acc, dst computes the set of atoms that existed on the source device src at the time m was sent. To this end, we first find the subset of messages in $msgs_{dst}$ that are referenced in the message dependencies $deps$:

$$\begin{aligned} \text{msgsIn}(deps) = \big\{(s, c, \_, \_, \_, \_, \_, \_) \in msgs_{dst} \mid \\ \exists\, c'.\ (s, c', \_) \in deps \wedge c \leq c'\big\} \end{aligned} \tag{5.20}$$

As defined in (5.10), the set of atoms at the time m was sent is the set of atoms that were inserted but not deleted in the set of messages $\text{msgsIn}(deps) \cup \{m\}$. Thus, dst can check that the accumulator satisfies:

$$acc \stackrel{?}{=} x_{src}^{P(S(\text{msgsIn}(deps) \cup \{m\}))r} \mod N_{src}. \tag{5.21}$$

If dst has already verified the message hash $\text{h}(m)$, this check is redundant and only serves to verify that src has calculated acc correctly. However, dst can only verify the hash if it can compute the hashes of all dependencies, which may not be the case if it does not know the full operation history because it has joined from a snapshot (as described in Section 5.4.3). If any dependencies of a message predate (or happened concurrently to) the snapshot from which dst was initialised, verifying the accumulator allows dst to check that the sender's state is consistent with its own.

If dst has already verified an earlier accumulator acc_{old} (with corresponding nonce r_{old}) from src, it can compute the new accumulator incrementally. Let S_{added} be the set of atoms added, and S_{removed} the set of atoms removed since acc_{old}. To add all atoms in S_{added}, dst can perform modular exponentiations to compute $acc_{\text{old}}^{P(S_{\text{added}})} \mod N_{src}$. As the factorization $N_{src} = p_{src} q_{src}$ is private to src, dst cannot compute multiplicative inverses modulo $\varphi(N_{src})$, and thus cannot remove elements from src's accumulator. However, dst

can check if the provided accumulator *acc* is correct by adding the removed atoms to *acc*:

$$acc_{\text{old}}^{P(S_{\text{added}})r} \stackrel{?}{=} acc^{P(S_{\text{removed}})r_{\text{old}}} \mod N_{src} \tag{5.22}$$

Lastly, the destination device verifies that *mh* has been computed correctly by recomputing the value based on its own operation hash trees.

5.4.3 Adding a new collaborator

Similarly to the process in Section 5.3.4, the device sending the invitation first broadcasts a message containing an addDevice(*deviceID*, *publicKey*) operation to the existing collaborators, where the public key now also contains the accumulator RSA modulus of the device. Next, the collaborator A who invites the new device sends a *snapshot* to the new device. The snapshot is a 4-tuple $(S_A, sdesc, mproofs, mnodes)$, where $S_A = S(msgs_A)$ is A's current set of atoms, as defined in (5.10). We show in Section 5.4.3 how *sdesc* is constructed and checked, and we discuss *mproofs* and *mnodes* in Section 5.4.3. Using the snapshot, a new device B can start collaborating from the current state, but does not learn contents that were added to the document earlier but deleted since then. Note that to ensure this privacy property, devices must not forward a message to devices that were added later (in the dependency graph) than the message. After B has received a snapshot, it immediately broadcasts a message containing a noop operation. The accumulator value of this message allows other devices to verify that B has received a set of atoms consistent with their own. Because B cannot verify the hashes of dependent messages that happened before or concurrently to a snapshot, it must only accept messages that happened after its noop message. If any messages happened concurrently to the snapshot, B must request a new snapshot that also contains the effects of these concurrent messages. After processing this snapshot, B publishes a new noop operation. Whether a message was created logically before, after, or concurrently to another message, can be determined straightforwardly based on the sequence numbers in the message and its dependencies.

State descriptors

sdesc is the set of *state descriptors*, one for each of the existing collaborators. A state descriptor is an 8-tuple $(src, ctr, hash, deps, acc, mh, sig, wit)$, where the first seven elements

are taken from the most recent message sent by *src*:

$$\begin{aligned} sdesc = \big\{ & (src,\ ctr, \mathrm{h}(m), deps, acc, mh, sig, wit) \mid \\ & m \in msgs_A \wedge \\ & m = (src, ctr, op, deps, r, acc, mh, sig) \wedge \\ & wit = \mathrm{witness}(S(\mathrm{msgsIn}(deps) \cup \{m\}), r) \wedge \\ & \nexists\, c'.\ ctr < c' \wedge (src, c', \_, \_, \_, \_, \_, \_) \in msgs_A \big\} \end{aligned} \tag{5.23}$$

The last element, *wit*, is a *witness* that cryptographically proves the relationship between *acc* (the accumulator from *src*) and S_A (the current set of atoms):

$$\mathrm{witness}(S_{src}, r) = x_{src}^{P(S_{src} - S_A) \cdot r} \mod N_{src} \tag{5.24}$$

Using (5.20), let $S_{src} = S(\mathrm{msgsIn}(deps) \cup \{m\})$ be the set of atoms in the document at the time when m, the most recent message from *src*, was sent. S_{src} may reflect the state of the document at some point arbitrarily far in the past, depending on the time when *src* was last active. If A knows the full message history, S_{src} is known to A. We will consider the case where A has only seen a partial history in Section 5.4.3. The newly invited collaborator, however, does not know S_{src} for any $src \neq A$, since the snapshot contains only S_A, the current set of atoms on device A.

In the intervening time between state S_{src} and the current state S_A, atoms may have been added or removed. The set $S_{src} - S_A$ in the exponent of (5.24) contains exactly those atoms that have been removed.

When device B receives a snapshot from device A, it performs the following steps to verify S_A and *sdesc*:

1. For each atom $(pos, src, ctr, txt) \in S_A$, verify that:

   (a) The pair (src, ctr) is unique:

   $$\forall p, t.\ (p, src, ctr, t) \in S_A \implies p = pos \wedge t = txt. \tag{5.25}$$

   (b) The atom's *ctr* is contained in the state descriptor for device *src*:

   $$\exists\, c'.\ (src, c', \_, \_, \_, \_, \_, \_) \in sdesc \wedge ctr \leq c'. \tag{5.26}$$

   (c) The atom's *ctr* is contained in *deps* in A's own state descriptor:

   $$\begin{aligned} \forall deps.\ (A, \_, \_, deps, \_, \_, \_) \in sdesc \implies \\ \exists\, c'.\ (src, c', \_) \in deps \wedge ctr \leq c'. \end{aligned} \tag{5.27}$$

2. For each state descriptor, i.e. for each $(src, ctr, hash, deps, acc, mh, sig, wit) \in sdesc$:

   (a) Verify that sig is a valid signature of $docID \parallel src \parallel ctr \parallel hash \parallel deps \parallel acc \parallel mh$, checked with src's public key.

   (b) Find the subset of atoms in S_A that already existed in S_{src}. Although the set S_{src} is not known to the newly invited device B, the intersection $S_{src} \cap S_A$ can be computed from src's state descriptor:

$$\begin{aligned} S_{src} \cap S_A = \big\{ & (p, s, c, t) \in S_A \mid \\ & (s = src \ \wedge \ c \leq ctr) \vee \\ & (s \neq src \ \wedge \ \exists\, c'.\ (s, c', \_) \in deps \ \wedge \ c \leq c') \big\} \end{aligned} \tag{5.28}$$

   We can then use wit to verify that the computed set $S_{src} \cap S_A$ is indeed a subset of S_{src}:

$$wit^{P(S_{src} \cap S_A)} \stackrel{?}{=} acc \mod N_{src}. \tag{5.29}$$

   If the snapshot is correct, the exponent from (5.24), $P(S_{src} - S_A) \cdot r$, is multiplied with the exponent $P(S_{src} \cap S_A)$ from (5.29), yielding $P(S_{src}) \cdot r$ as in the accumulator definition (5.11).

   (c) Check that ctr is the most recent sequence number seen from src:

$$\begin{aligned} \forall d.\ (\_, \_, \_, d, \_, \_, \_) \in sdesc \implies \\ \forall c.\ (src, c, \_) \in d \implies c \leq ctr. \end{aligned} \tag{5.30}$$

   (d) Ensure that there is a state descriptor for every device in $deps$:

$$\forall s.\ (s, \_, \_) \in deps \implies (s, \_, \_, \_, \_, \_, \_) \in sdesc. \tag{5.31}$$

If any of the above checks fail, the snapshot must be rejected.

Computing witnesses incrementally

The above discussion, especially (5.23) and (5.24), assumes that the device A that sends the snapshot has access to the full message history since the creation of the document. In general, this may not be the case, since A might itself be a device that was invited by snapshot.

However, the approach above easily generalises to the case where A starts from a snapshot. In particular, the witness computation in (5.24) can be performed incrementally without knowledge of S_{src}. Notice that the exponent $P(S_{src} - S_A)$ contains those atoms that existed in S_{src} but have been removed in S_A. Thus, device A can start with the witness

$wit = x_{src}^{P(S_{src}-S_A)r} \mod N_{src}$ for *src*, which is included in the snapshot. Subsequently, whenever an atom a is removed from S_A, and $a \in S_{src} \cap S_A$, the witness needs to be updated by computing $wit' = wit^{\text{prime}(a)} \mod N_{src}$. Adding an atom to S_A, or removing an atom that is not in $S_{src} \cap S_A$, leaves the witness unchanged. When A receives a new message from *src*, it needs to recompute the witness from scratch. For this, A needs to determine all atoms that have been deleted since that message and perform a modular exponentiation for each atom. However, we expect that in most cases in practice if *src* is online, the latest message will correspond to a state equal or close to the most recent state at A, in which case that set is empty or small.

When A needs to construct a snapshot to invite a new device, they can then use the latest witness for each *src*. Witness computation may be delayed until the time a snapshot is generated, however it may be desirable to update the witnesses incrementally with every message received or periodically, in order to allow snapshots to be created quickly.

Merkle tree consistency proofs

The third element of the snapshot, *mproofs*, serves as a cryptographic proof that there has not been a *fork* in the editing history of the document. A fork occurs if a device presents different and contradictory edits with the same sequence number to its collaborators.

In the basic protocol of Section 5.3, the message hashes in *deps* serve the purpose of detecting forks. In the privacy-enhanced protocol of Section 5.4, the full message history is not available to a newly invited collaborator, so we instead use Merkle trees to prove that there is no fork among the state descriptors in *sdesc*.

As described in Section 5.4.1, each device keeps track of the sequence of messages it has received from each collaborating device – an append-only log per source device. Following the approach of Certificate Transparency [CW09, LLK13], we encode that log in a Merkle tree. If no fork has occurred, each device will see the same sequence of messages from each source device. However, since devices may be offline, some devices may have an incomplete view of other devices' message logs. In those cases, we expect the message log on one device to be a prefix of the corresponding logs on other devices.

We use Merkle consistency proofs to show that the per-source message sequence on one device is a prefix of the corresponding message sequence on another device, without revealing the actual messages. For each originating device *src*, *mproofs* contains a set of Merkle consistency proofs as follows. Let $T_{src}[c]$ be the Merkle tree containing the first c messages from *src*. Let c_d be the sequence number of the last message from *src* seen by d at the time of the message corresponding to d's state descriptor. Then $T_{src}[c_d]$ contains exactly the messages from *src* received at device d at that time.

Now consider the Merkle trees $T_{src}[c_d]$ for all devices d, sorted in increasing order by c_d, and omitting duplicate c_d. For each adjacent pair of Merkle trees in this set, *mproofs*

contains a consistency proof showing that the larger tree is the same as the smaller one with some additional leaves appended. By transitivity, those proofs show the consistency of all trees for each originating device.

To check the consistency proofs, we proceed as follows. From each state descriptor, extract the sequence number of the last received message from each device *src*. Group the sequence numbers by originating device *src* and sort them in increasing order, omitting duplicates. Now, for each two adjacent sequence numbers c_1, c_2 in this list, check that *mproofs* contains a valid consistency proof between Merkle trees $T_{src}[c_1]$ and $T_{src}[c_2]$, i.e. a proof that $T_{src}[c_1]$ is a prefix of $T_{src}[c_2]$. The Merkle tree roots do not need to be included with the proofs if they can be computed from a matching consistency proof. Compute the Merkle tree roots of the trees $T_{src}[c]$ for each counter c, and using those, verify the hash over the Merkle tree roots *mh* within each state descriptor.

Lastly, *mnodes* contains a partial Merkle tree for each device, containing all nodes of the latest tree T_{src} that are required for the newly invited collaborator to construct its own Merkle trees. These partial trees need to contain enough information that the new collaborator can extend the tree by appending leaves, for which it is sufficient to include the root of every maximal complete subtree.

5.5 Related work

As discussed in Section 2.1, traditional **collaborative editing** applications rely on OT algorithms [NCDL95, EG89] to synchronize changes between devices. More recently, (operation-based) CRDTs [SPBZ11b, SPBZ11a] have been proposed to ensure convergence without requiring consensus between devices, providing strong eventual consistency. At the time of writing, there are a number of projects actively working on collaborative editors or libraries based on CRDTs that allow devices to communicate peer-to-peer (using WebRTC), including Teletype for Atom, Conclave, and Automerge. A number of authors have also proposed server-based collaborative editing systems that provide end-to-end encryption such as SPORC [FZFF10], SECRET [FMMS17], or Capsule [Kob18]. However, to the best of our knowledge, our protocol is the first one that provides authenticated snapshots and allows devices to verify that their view is consistent with other devices, even when other devices are offline.

Version control systems (Section 2.1.3) such as Git, Mercurial, or Subversion are another popular kind of tool for collaboration. They are not designed for real-time editing and require manual merging if a possible conflict is detected. Authenticated snapshots could also be implemented for version control systems; however we are not aware of any existing system that supports them.

Cryptographic accumulator schemes (Section 2.3) have been proposed based on RSA [BP97, BD93], bilinear maps [Ngu05, CKS09], Merkle trees [CHKO12], and vector commitments [CF13]. Variants of our protocol could also be designed based on other accumulator schemes. However, we found RSA accumulators especially suitable because they provide constant size public keys, witnesses, and batch membership proofs.

In a three-party **authenticated data structure** (ADS) [Tam03], a *source* replicates some data to one or more *servers*, and the servers answer queries on the data from *clients*, including a proof that allows clients to verify the authenticity of the response using a *digest* provided by the source (e.g. a hash). Our proposed scheme can be seen as an ADS for CRDTs, where the collaborators are sources, the inviter is the server, and a newly invited device is the client.

A **redactable signature** scheme (Section 2.4) allows a third party without knowledge of the secret signing key to remove parts of a signed message while still retaining a valid signature. Our protocol essentially uses redactable signatures – the signature within a message signs the current set of atoms, and the state descriptors within a snapshots contain a signature of a possibly redacted state of that set.

5.6 Summary

In this chapter, we proposed a protocol for peer-to-peer collaborative editing that allows new devices to be added as collaborators by sending a snapshot that only contains the latest state of a document. Such a snapshot reduces the amount of data that needs to be transferred to a new device and additionally hides the editing history of the document, while still allowing the new device to verify its integrity. This is achieved without requiring a consensus between collaborating devices and is therefore also suitable for devices that are frequently offline.

Future research might look at protocols that preserve information about the positions where text fragments have been deleted, or alternatively, completely hide this information. Another interesting research direction is developing CRDTs specifically designed for authenticated snapshots and history privacy, with a reduced overhead. It would also be interesting to design a protocol that does not only hide deleted parts from a new user, but also hides the author of a piece of text, either from new users or from all collaborators.

In the next chapter, we will evaluate the performance of the protocol in terms of its overhead and to what extent it meets the design goals stated in Section 5.2.1.

CHAPTER 6

AUTHENTICATED SNAPSHOTS PROTOCOL EVALUATION

In Chapter 5, we described a protocol for collaborative editing with authenticated snapshots. In this chapter, we evaluate the costs of the protocol, and how well it reaches the design goals described in Section 5.2.1.

Section 6.1 provides an empirical evaluation of the computational and communication costs, the memory and storage requirements, and an analysis of the scalability of the protocol. Section 6.2 explains how edit integrity and attributability are achieved, and Section 6.3 discusses edit history privacy. Section 6.4 shows how the protocol achieves the desired consistency properties by providing a proof of how consistency between honest devices is preserved even in the presence of arbitrarily misbehaving devices. Section 6.5 considers convergence and availability.

6.1 Costs and scalability

Significant costs arise for the creation and processing of messages, for inviting a new collaborator, and for joining as a collaborator. We consider the computational costs of these actions, the communication costs for different types of messages, and the memory and storage requirements.

We implemented a prototype of the privacy-enhanced protocol in Java based on the Treedoc CRDT with unique disambiguators [PMSL09], without optimizations. The instrumented prototype simulates all devices within a single thread of execution and measures execution times of relevant operations as well as the volume of network communication. We use a 2048-bit RSA modulus, and SHA-256 as the secure hash function.

To evaluate the costs of the scheme based on realistic data, we replayed edits from Wikipedia editing histories. We randomly[1] selected 300 pages from the English Wikipedia on 16 May 2018. We excluded seven pages with only a single edit, and to ensure a reasonable emulation time, we excluded 23 pages which had either more than 250 edits, or more than 25,000 characters in the latest version; our results in this section demonstrate clear trends which will not significantly change for larger edit histories or pages.

For the sake of estimating the communication costs, we assumed that *deviceID*s are 128-bit random numbers (to achieve uniqueness with high probability in a decentralized setting). For simplicity, we assume that all devices are always online, devices do not batch multiple operations together into a single message, and that devices only send messages when they edit the document. When replaying the editing history, we assumed that a Wikipedia user or IP address corresponds to a device, and that new collaborators get invited by and receive a snapshot of the document from the last person who edited the document before them. We assume each line is represented as an atom, as commonly done in the evaluation of CRDT algorithms [WUM09, WUM10, NMMD13] (the original Treedoc paper used paragraph-level granularity [PMSL09]).

We did not consider the time taken for encrypting or decrypting messages, since the choice of the encryption scheme is independent of our protocol, and modern encryption algorithms are fast compared to the RSA accumulator operations. For signing, we used ECDSA and the NIST P-256 curve.

We measured execution times on a 2013 desktop-class 3.20GHz i5-4570 CPU with 32 GiB RAM running Oracle JRE 1.8.0_172 with a heap size of 8 GiB. We chose a 8 GiB heap size to reduce the number of garbage collection cycles and their impact on the measurements, and because we simulated all devices within a single process. The heap size of 8 GiB was enough to comfortably simulate up to 141 devices, therefore a single device can run the protocol with substantially less memory.

The source code used for the experiments is available [Kol19]. The Appendix contains the list of randomly selected Wikipedia pages used.

Prime representative generation Since RSA accumulators require a hash function that produces primes for collision-freeness, we implement the prime representative generator based on a method by Barić and Pfitzmann [BP97] as described in Section 2.3.1: For an input x, we compute $\hat{h}(x) = \text{SHA-256}(x)$, and find the smallest t-bit number d such that appending d to $\hat{h}(x)$ results in a prime number. In other words, we keep incrementing d until $2^t\hat{h}(x)+d$ is prime. We test numbers for primality using the Miller-Rabin probabilistic primality test [Rab80] with 50 iterations. We chose $t = 16$, since assuming Firoozbakht's

[1] We used `https://en.wikipedia.org/wiki/Special:Random`. For the curious, the selection process is not uniform. Each Wikipedia page is assigned a "random index" from the interval $[0, 1)$, and to select a page, another random number is drawn from $[0, 1)$ and the page with the next higher index is returned.

conjecture [Rib04, p. 185] (which implies that the gap between primes p_k and p_{k+1} is less than $\ln^2 p_k - \ln p_k$ for all $k > 4$ [Sin10]), this should always allow a suitable d to be found.

Discussion of simplifying assumptions In a practical implementation, devices may want to batch edits, and periodically broadcast noop messages to other clients to confirm the latest seen document state. Devices may also be offline temporarily or permanently, delaying message delivery and processing until such devices comes back online again, but this merely defers when costs are incurred.

Periodically broadcasting noop messages would cause additional network traffic and devices would need to process additional messages. The dominant cost for processing noop messages is the verification of the accumulator of the sending device (see Section 6.1.1.1). This cost grows linearly with the number of atoms added and deleted since the last message from the device. Therefore, processing noop messages would reduce the computational cost for the accumulator verification for individual operations, however it is likely to increase the cumulative cost if a large number of atoms are added, and the same ones deleted, between edit operations from a device. Some additional costs may also be caused by devices that regularly send noop messages, but do not make any (more) edits. On the other hand, for devices that regularly send noop messages, other devices can skip the iterative witness computation, as the witness is simply the accumulator base if a device is up-to-date. Since we do not have reliable data on the network status of devices editing Wikipedia, we defer evaluation of these trade-offs to future work.

6.1.1 Computation costs

The following describes the computational costs of processing individual operations. We use c for the current number of collaborating devices, n for the current number of atoms in the document, and m for the total number of distinct messages broadcast since the document was created.

6.1.1.1 Basic editing operations

Processing any message requires checking the correctness of the hashes and the accumulator. Of those, verifying the accumulator tends to be the most costly, as it requires a modular exponentiation for every atom added or deleted since the last message from the source device.

Insert operation In addition to the above, the dominant costs for inserting an atom are calculating a prime representative, and updating the device's accumulator, which requires one modular exponentiation. Figure 6.1 shows the processing times we measured for insert operations. The time does not grow with the number of devices, the median processing

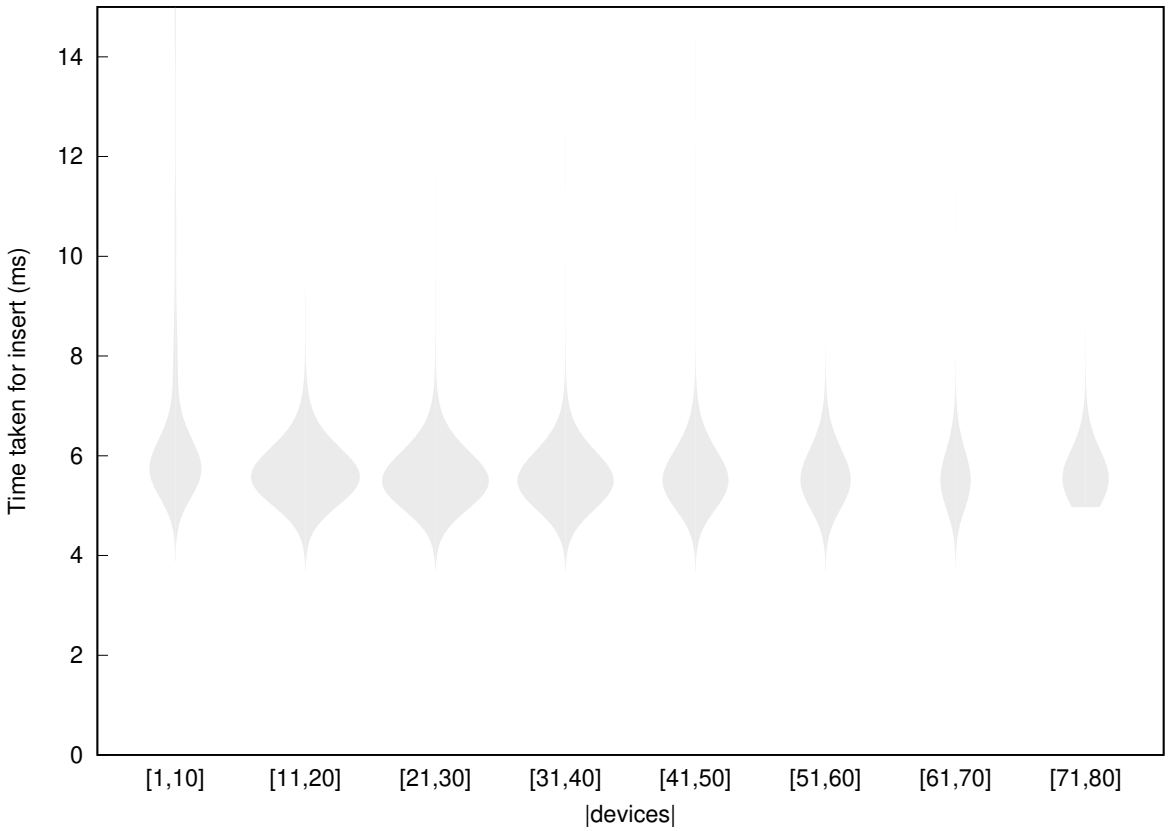


Figure 6.1: Measured processing times for an insert operation from another device. The time is relatively constant in most cases, but can vary significantly for individual messages depending on the number of changes that have happened since the last message from the same device because verifying the updated accumulator takes time linear in the number of changes. The plot does not show data for more than 80 collaborators due to lack of representative data; only 6 out of 270 pages had more than 80 collaborators in total. Overall, the median processing time is 5.6 ms, and the 99^{th} percentile is 11.0 ms.

time for a message containing an insert operation from another device was 5.6 ms, and 99% were processed within 11.0 ms. We observed outliers of up to 1.0 seconds, which were caused by the cost for verification of the accumulator when a relatively large number of changes have happened since the last message from the device that created the insert operation. Note that these numbers are only for insert operations created on a different device. Processing locally generated operations is faster, since they do not require the source device's accumulator to be verified.

Delete operation For a delete operation, the additional costs are dominated by the cost of updating the device's accumulator, which requires a modular k-th root computation, and by the cost for the iterative witness computation. Figure 6.2 shows the time it took to process delete operations from other devices, with iterative witness computation enabled after every operation. Overall, the median cost for processing a delete operation was 12.4 ms, and 99% were processed within 64.9 ms. Outliers take up to 1.19 seconds and

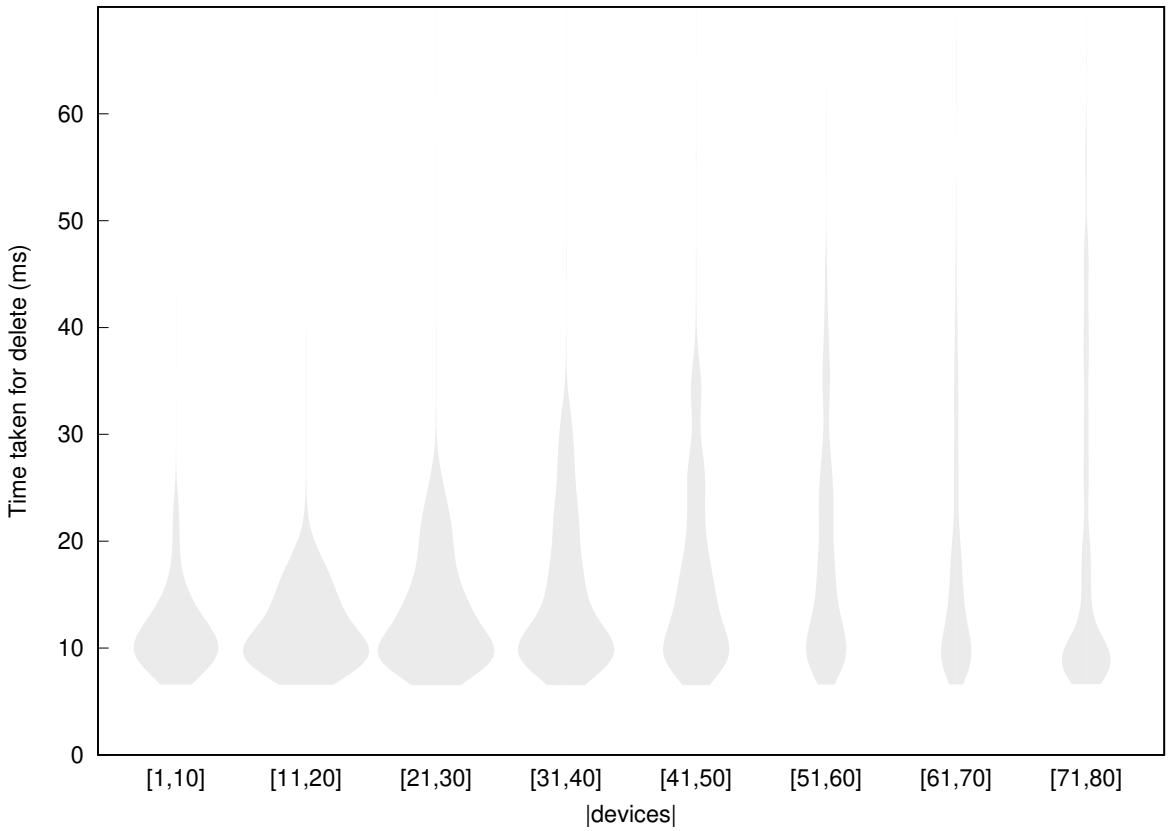


Figure 6.2: Measured processing times for a delete operation from another device. The worst case execution time generally grows with the number of devices, as with every delete operation, we incrementally update the witnesses for all other devices, unless the deleted atom has been inserted after the last message from a device. Therefore, costs vary depending on the editing behaviour of users, and how often they communicate. Roughly speaking, deleting older parts of a document is more expensive, and more frequent synchronisation between devices also makes deletions more expensive. We omit data for more than 80 devices where we have limited data. Overall, the median processing time is 12.4 ms, and the 99^{th} percentile is 64.9 ms.

were due to the accumulator verification. As for the insert operation above, these numbers do not consider delete operations that were created on the same device, which are faster to process.

6.1.1.2 Adding a new collaborator

Adding a new collaborator requires four steps:

1. Generating a snapshot on the inviting device,
2. verifying the snapshot and the new device,
3. initialising the local state on the new device, and
4. verifying the first message from every other device on the new device, and vice versa.

Snapshot generation Creating a snapshot requires $\mathcal{O}(n + c)$ (simple) operations, computing $\mathcal{O}(c^2)$ Merkle consistency proofs, plus computing a witness per device. Computing the consistency proofs is fast for a moderate number of devices. Computing the witness for a device requires a modular exponentiation for every atom that has been deleted since the last accumulator seen from that device (but was already present then). However, we iteratively compute witnesses with every message to minimize snapshot generation time, as described in Section 5.4.3. Therefore, the time taken to generate a snapshot is negligible compared to other costs such as its verification.

Snapshot verification Verifying the Merkle consistency proofs can take $\Theta(c^2 \log m)$ time, where m is the total number of distinct messages broadcast since the document was created. However, in practice, the cost for verifying a snapshot is dominated by the costs for verifying that the set of atoms matches the accumulators, unless the number of collaborators becomes large compared to the number of atoms in the document, and many of them have sent their last message at different points in the history. For each device, this requires one modular exponentiation per atom that is present both in the latest document and in the accumulator from the device. Thus, for moderately large groups of collaborators, the worst case cost is $\mathcal{O}(c \cdot n)$. The actual cost for a device is significantly smaller if a large number of atoms have been added since the last message the device. Figure 6.3 shows how long each snapshot verification took in our experiments, and its relationship to the product of the number of atoms and devices.

Initialisation from snapshot After a snapshot is verified, the remaining cost for initialising a new device is dominated by the cost for calculating the device's current accumulator value based on the current set of atoms. The cost is $n - 1$ modular multiplications and a single modular exponentiation.

Verifying first message from device When a device receives the first message from another device, it needs to compute the current set of atoms at that device based on the counters within the message and the operation history, and based on that verify the accumulator value by re-computing it. This requires $O(n)$ modular exponentiations. We empirically verified this linear relationship; we observed a verification time of about 0.65 milliseconds per atom. Figure 6.4 shows the execution times of the verification for our experiments with the Wikipedia pages; it demonstrates the linear dependency of the execution time on the number of atoms present in the document at the time of the message.

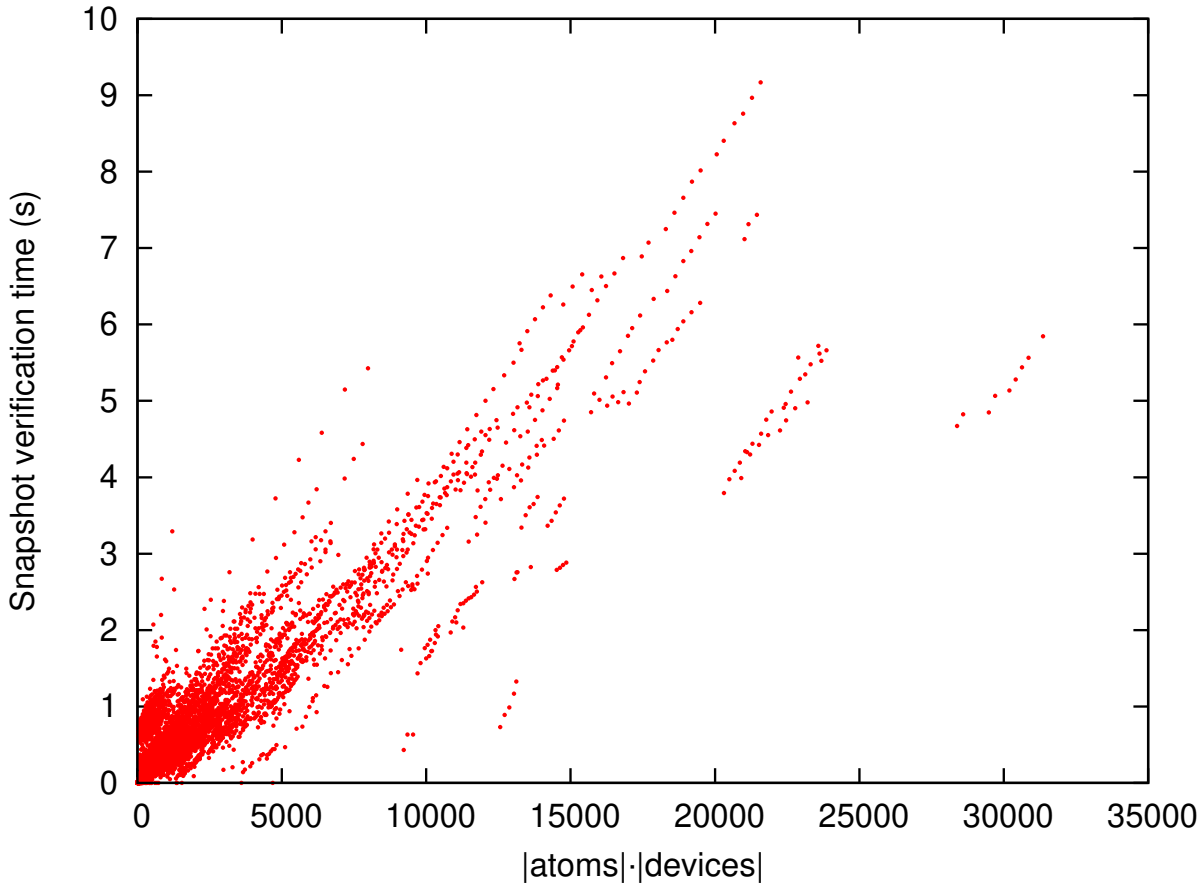


Figure 6.3: Measured snapshot verification times. Verification time is at most linear in the product of the number of devices and atoms, however it can be significantly lower if many atoms have been added since the last message from collaborating devices.

6.1.2 Communication costs

Table 6.2 compares the amount of data transferred for different message types for the basic and the privacy-enhanced protocols. The privacy-enhanced protocol requires additional data for individual messages (for the nonce and accumulator value), but snapshot sizes are smaller if the number of users is small compared to the number of atoms, since deleted atoms do not need to be transferred. Using the Wikipedia data, we looked at the amount of data that would need to be transferred to invite the user that has most recently made her first contribution. Figure 6.5 shows a comparison of the amount of data transferred in the basic scheme and the privacy-enhanced scheme. We observed a median 84% reduction in data transferred for the privacy-enhanced scheme compared to the basic scheme. The reduction was always more than 30%, and 98.2% in the best case.

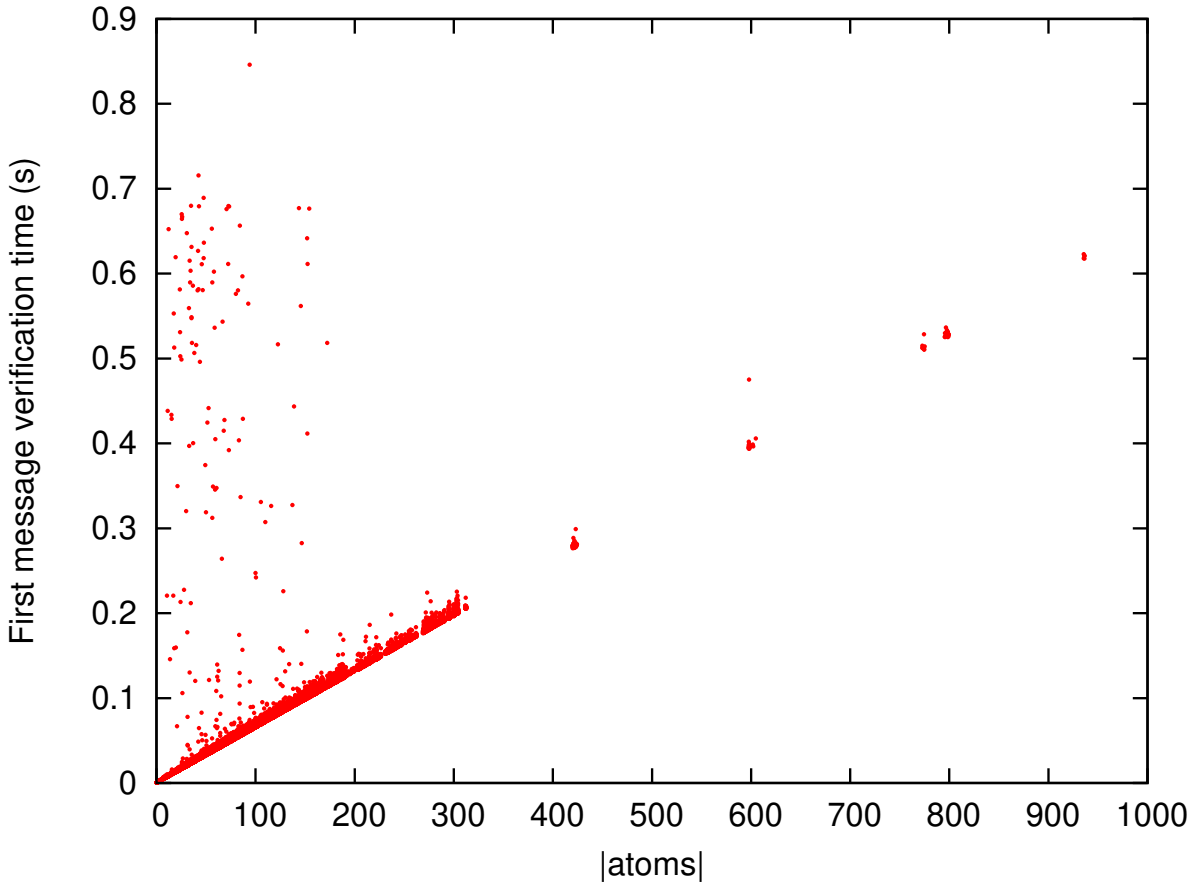


Figure 6.4: Measured execution times for verifying the first message from a device, dependent on the number of atoms within the document at the time of the message. The time depends largely (linearly) on the number of atoms because verification requires all atoms to be added to an RSA accumulator.

Variable	Description	Typical value	Empirical values		
			min	median	max
d	Number of collaborating devices	variable	2	16	141
s_{devID}	Size of device identifier	16 B			
s_{hash}	Size of hash	32 B			
s_{sig}	Size of signature	72 B			
s_{nonce}	Size of nonce	16 B			
s_{RSA}	Size of accumulator	256 B			
s_{pos}	Size of position identifier	variable	1 B	3 B	117 B
s_{content}	Size of atom text fragment	variable	1 B	34 B	5.0 KiB
s_{pubkey}	Size of public key (accumulator + signing)	256+32 B			
s_{history}	Size of message history (excl. signatures)	variable	4.9 KiB	123 KiB	7.2 MiB
s_{doc}	Size of document including metadata	variable	699 B	4.8 KiB	77 KiB

Table 6.1: Description of different variables used in Table 6.2, and typical values. For the ones where typical values are highly variable, minimum, median, and maximum values from our simulations with Wikipedia edit histories are included.

	Basic protocol	Privacy-enhanced protocol
Message	$s_{\text{devID}} + d(s_{\text{devID}} + s_{\text{hash}}) + s_{\text{sig}}$+ Op	Basic + s_{nonce} + s_{RSA} + s_{hash}
Op_insert	Message + s_{pos} + s_{content}	Basic
Op_delete	Message + s_{devID}	Basic
Op_noop	Message	Basic
Op_addDevice	Message + s_{devID}	Basic + s_{pubkey}
Snapshot	$s_{\text{history}} + d \cdot s_{\text{sig}}$	$s_{\text{doc}} + d \cdot (s_{\text{devID}} + s_{\text{hash}} + 2s_{\text{RSA}} + \mathcal{O}(d \cdot \log s_{\text{history}}) + s_{\text{sig}})$

Table 6.2: Communication costs for different types of messages and operations. Small constants are omitted. Note that in the basic protocol, for a snapshot it is sufficient to include the most recent signature from each device.

6.1.3 Storage and memory requirements

A device needs to keep the atoms currently in the document in memory. In addition, it must store, for each collaborator, the most recent message, the current witness, and additional metadata. The device needs to store the message history to be able to relay messages to other devices, and to calculate earlier states of the document which can be needed to verify an accumulator or to calculate a witness. The memory requirements for storing current atoms in a document corresponds to the y-axis in Figure 6.5, and the past history corresponds to the x-axis. Therefore, the overall memory and storage requirements are typically less than 10 MiB. A device may also keep an in-memory or disk cache of the prime representatives of all atoms (34 bytes per atom in our prototype) as computing those is costly. If memory/storage is scarce and the prime representative generator described in Section 2.3.1 is used, it can also memorize only the last 2 bytes, and recompute the remaining ones when needed.

6.2 Edit integrity and attributability

Cryptographic signatures attached to each message ensure that only group members can modify the document. For the basic protocol of Section 5.3, the signatures also provide attributability of all modifications. When using the privacy-enhanced protocol of Section 5.4, a device can similarly use the signatures to attribute any changes made to the document after it joined. Parts of the document that were added earlier – before or concurrently to when a device joined – cannot be attributed directly using the signatures of the messages containing the insert operations, since the new collaborator does not receive those messages. However, attributability in this case is ensured by the signed accumulators from each collaborator that are part of each snapshot, since the set of atoms certified by each device in this way must also contain all atoms inserted by the device itself.

6.3 Edit history privacy

In the privacy-enhanced protocol, a new device, when joining, only receives the current set of atoms, the set of devices collaborating, several sequence numbers and cryptographic hashes, and a set of RSA accumulators. Assuming preimage resistance of the hash function and due to including a 128-bit random nonce into every message, it is infeasible to infer anything about previous contents from the hashes. RSA accumulator and witness values each contain an accumulated 128-bit random nonce; since the new device never learns this nonce, the accumulators are cfw-indistinguishable [DHS15], making it infeasible to infer contents from the accumulator values. For efficiency reasons, we use the same nonces to

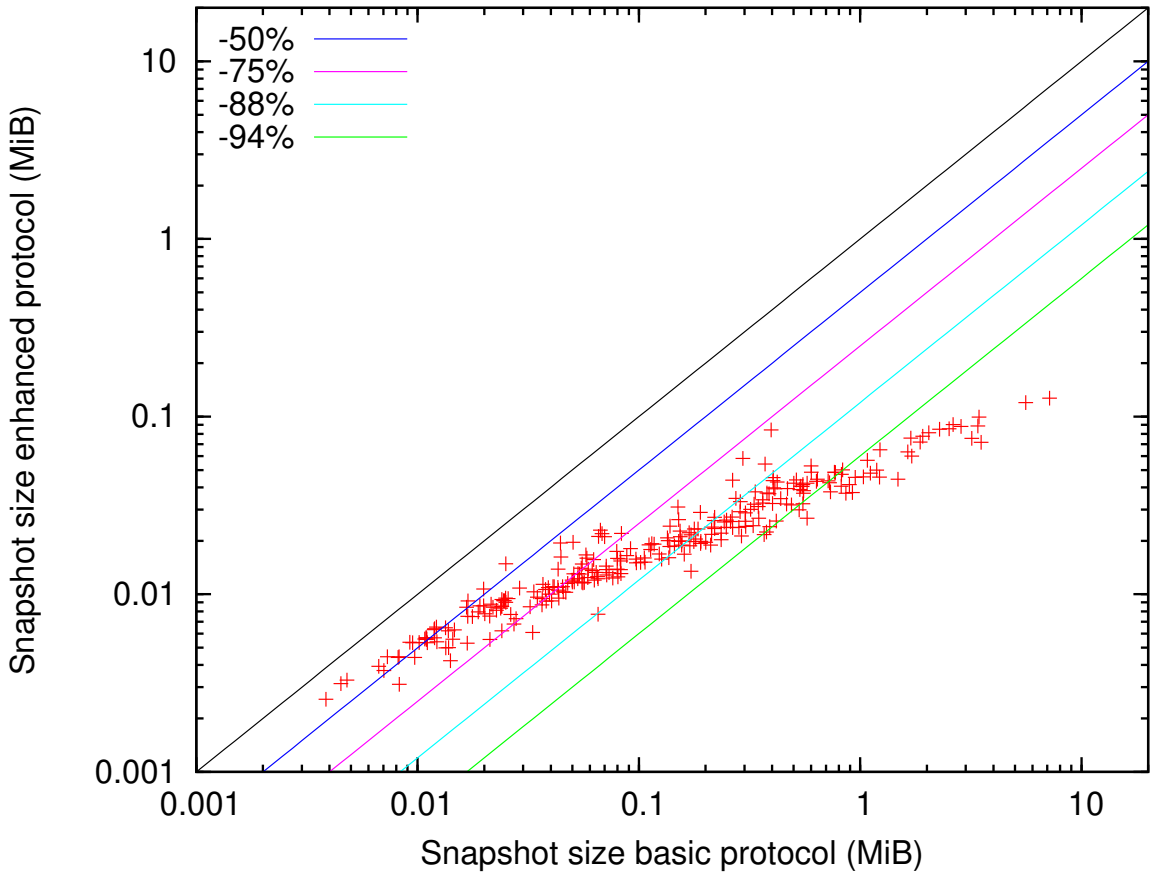


Figure 6.5: Amount of data transferred to a new collaborator when invited by another device, for the most recently added collaborator in each of the pages from the Wikipedia dataset for the privacy-enhanced protocol in comparison to the basic protocol. The plot is log-log scale. A point below the black diagonal line indicates that the privacy-enhanced protocol transfers less data than the basic protocol. The privacy-enhanced scheme needs to transfer less data because it does not transfer deleted atoms.

calculate message hashes and accumulators to improve efficiency; we believe this does not introduce any weaknesses.

While the scheme hides the contents of all text deleted before a device joins, it does not perfectly hide the editing history. Since a snapshot also includes metadata such as position identifiers and sequence numbers, a new device can infer some information about the history, such as the number of messages sent by each device. Moreover, gaps between position identifiers can leak the fact that atoms have been deleted at a certain position (but not the values of those atoms).

6.4 Consistency and snapshot consistency

We show that our protocol satisfies a variant of *fork-join-causal consistency*, as introduced by Mahajan et al. [MAD11, MSL+11]. Stated informally, this consistency model requires

that honest[2] devices always observe the system in a state that is consistent with a global execution graph, and that this execution graph correctly reflects the dependencies and operations performed by devices.

To prove that our protocol satisfies this consistency model, we first show how to construct the *happens-before* graph G (representing the global execution). For each honest device n we also define a graph G_n representing n's view of the execution. We then prove that G and G_n are consistent with each other: that is, reading the document at a vertex of G_n returns the same result as reading it at the corresponding vertex of G.

Definition 6.1. Let G be a directed acyclic graph. We then define the partial order $\prec_G$ to be equal to the transitive closure of the graph. That is, for vertices a and b in G, we have $a \prec_G b$ if there is an edge $a \to b$ in G, or if there exists a vertex c such that $a \prec_G c$ and $c \prec_G b$. Similarly, the partial order $\prec_{G_n}$ is defined as the transitive closure of the graph G_n.

Definition 6.2. An *operation message* is a message containing an operation (insert, delete, noop, or addDevice, but not a snapshot) sent as part of the protocol.

Definition 6.3. For any vertex m in the graph G we define $\mathit{read}(G, m)$ to be the set of atoms in the document at the time immediately after m has been processed, i.e. the set of atoms a such that there exists a vertex $m_{I,a}$ containing an insert operation for a, with $m_{I,a} \preceq_G m$, and there exists no vertex $m_{D,a}$ containing a delete operation for a with $m_{D,a} \preceq_G m$:

$$\mathit{read}(G, m) = S(\{m' \mid m' \preceq_G m\}) \tag{6.1}$$

where the function $S(\dots)$ is defined in (5.10).

We can now formally define the fork-join-causal consistency model as follows.

Definition 6.4. An execution is fork-join-causally consistent if there exists a directed acyclic graph G (the *happens-before graph*) that satisfies the following three properties:

FJC0. G contains a vertex for every operation message sent by an honest device, and also a vertex for every operation message that is sent by a faulty device and processed by at least one honest device.

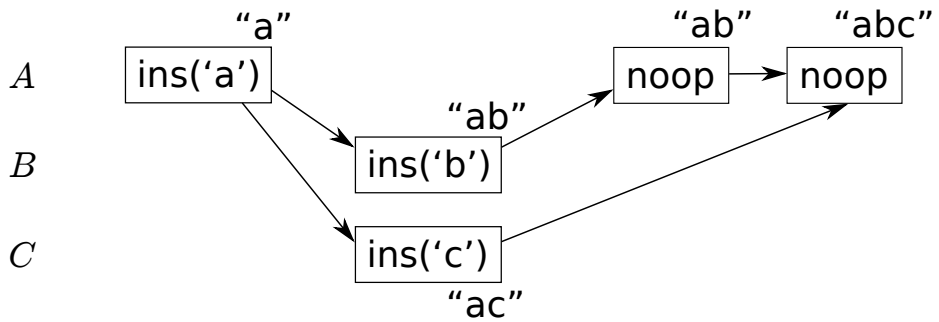
FJC1. The operations of an honest device are totally ordered in G. This total ordering must be consistent with the actual execution order of the operations at that device. Specifically, if v and v' are operations by n, then $v.\mathit{startTime} < v'.\mathit{startTime} \iff v \prec_G v'$.

[2]We use the word "honest" to refer to devices that correctly follow the protocol (in the distributed systems literature, the term "correct" is more common). A device that does not correctly follow the protocol, regardless whether by accident or by malice, is called "faulty".

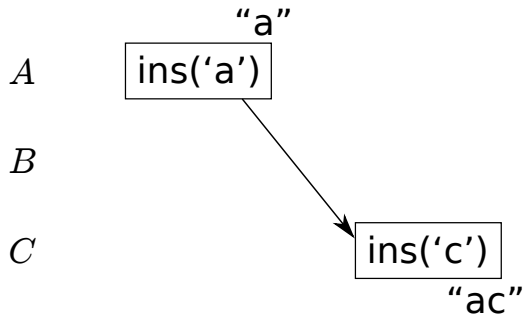
FJC2. For each honest device n there exists a directed acyclic graph G_n in which there is a vertex for every operation message sent or received by n, and edges corresponding to the dependencies between those messages. By FJC0, for each vertex m in G_n there is a corresponding vertex m in G. We then require that for each vertex m in G_n, the document state is the same as the document state at the corresponding vertex in G: $read(G_n, m) = read(G, m)$.

Basic protocol For the basic protocol described in Section 5.3, G can simply be defined as follows: G contains a vertex for each message m sent or observed by an honest device, and a directed edge $a \to b$ between vertices a and b if a is one of the dependencies of b.

Figure 6.6a shows an example of a happens-before graph for an execution where device A inserts the atom 'a', followed by devices B and C concurrently adding atoms 'b' and 'c', respectively. Device A then performs noop operations in order to acknowledge the receipt of the edits from B and C. Figure 6.6b shows device C's view of the execution. For clarity, we omit addDevice operations.



(a) Happens-before graph of execution (G)



(b) Happens-before graph of C's view of the execution (G_C)

Figure 6.6: Happens-before graphs for an execution with three devices where devices B and C perform concurrent inserts.

This definition of G trivially satisfies property FJC0. Moreover, from the protocol definition it is relatively easy to see that property FJC1 is also satisfied. Every honest device increments its sequence number with every message it sends, and an honest device would not process a message from a device a that depends on a message from a with a

higher or equal sequence number. Hence, the messages sent by an honest device are totally ordered in G.

For the basic protocol, it is also easy to see that property FJC2 is fulfilled. If a device n processes a message m, it needs to have processed all messages that happened before m in G_n. The use of cryptographic hashes within the message dependencies ensures that the set of messages preceding m in G_n is the same as the set of messages preceding m in G.

Privacy-enhanced protocol In the privacy-enhanced protocol of Section 5.4, the above argument no longer works, since it relies on devices having received all messages that happened before a message m before processing m, which is not necessarily the case: devices do not receive messages that happened before they were added as a collaborator.

We illustrate the challenge by giving an example before proceeding to the formalisation. Consider the execution visualised in Figure 6.7.

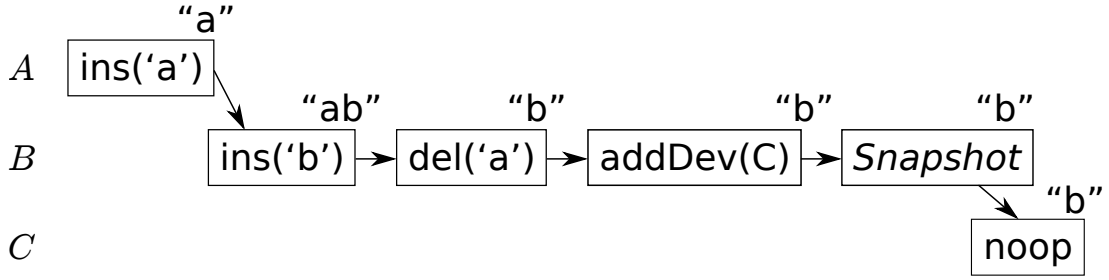


Figure 6.7: Happens-before graph of an execution where device B adds device C by sending it a snapshot of the current state.

Suppose that honest device A fails permanently after sending the insert operation for 'a', and therefore it never receives the later operations. Further assume that device B is faulty. Thus, there is no honest device that has observed the insert operation for 'b' or the delete operation for 'a'. Therefore it is not immediately clear how FJC2 can still be preserved for the noop operation by C (and any later operations).

One option would be to add a vertex containing an insert operation for each atom that C receives as part of the snapshot. However, this would allow too many executions. We only want to allow executions where snapshots are consistent with earlier messages seen by honest devices.

Since the messages containing the insertion of 'b' and the deletion of 'c' have not been observed by any honest device, it is not relevant for G whether they actually happened. It is only important whether it is possible to add a set of edit operations by faulty devices directly before the message adding a new device (or sending an updated snapshot) such that FJC1 and FJC2 are preserved. Thus we adapt the definition of G to allow the addition of vertices containing insert and delete operations by faulty devices between the vertices corresponding to messages observed by honest devices, and the vertex corresponding to the addDevice message for a new device, or a message containing an updated snapshot.

For each honest device n, Definition 6.5 describes the graph G_n that represents n's view of the execution. In summary, it contains insert operations for all atoms received by n in its initial snapshot, all messages processed by n, and edges for the dependencies between them. A device may receive more than one snapshot if another device performed operations concurrently to the first snapshot (as described in Section 5.4.3); if this is the case, n also contains insert and/or delete operations for atoms that were added/removed in subsequent snapshots.

Definition 6.5. For the privacy-enhanced protocol, we define G_n such that it contains:

1. A vertex for each operation message sent or processed by n.
2. An edge $a \to b$ between two messages processed by n if a is a dependency of b.
3. For each snapshot processed by n, a vertex r_i $(i = 1, ..., k)$. If n has sent any messages after the snapshot, add an edge $r_i \to t_i$ to the vertex t_i corresponding to the first such message.
4. If n joined as a collaborator from a snapshot, for each atom a that was part of this first snapshot, a vertex u_a with an operation insert(a),
5. For each subsequent snapshot received by n, a vertex u_a with an operation insert(a) for each atom present in the snapshot if there is no *previous* vertex with an insert operation for a in G_n. In this context, *previous* means preceding a vertex t_i corresponding to a vertex corresponding to n's first message after the snapshot.
6. Similarly for each subsequent snapshot received by n, a vertex w_a containing an operation delete(a) for any atom a with an insert operation, but no delete operation, previously present in G_n that is not present in the snapshot.
7. For each such vertex u_a or w_a, an edge to the vertex r_i corresponding to the snapshot.

Proof overview We construct a suitable happens-before graph G for an arbitrary execution, showing that G is a directed acyclic graph (Lemma 6.4.1), and hence that FJC0 and FJC1 are satisfied. Next, we show that deleted atoms cannot be re-added (Corollary 6.4.2.1), a property that is useful for Lemma 6.4.3, which shows that FJC2 holds for every message m, as long as it holds for every preceding snapshot. Finally, Lemma 6.4.4 shows that it holds for every snapshot, from which Corollary 6.4.4.1 deduces that FJC2 holds for G. We therefore conclude that the privacy-enhanced protocol is fork-join-causally consistent.

Definition 6.6. For a message $m = (src, ctr, op, deps, r, acc, mh, sig)$, let $v(m)$ be its version vector, containing the set of pairs (src_v, ctr_v) with the counter from the latest

message from each device included in m, i.e. $ctr_v = ctr$ for $src_v = src$, and otherwise ctr_v is the counter of the entry for src_v in $deps$. The version vector of a snapshot is defined equivalently using the state descriptor of the creator of the snapshot.

Definition 6.7. We construct the happens-before graph G for an execution as follows.

1. Add a vertex for each operation message sent or processed by at least one honest device.

2. Add a directed edge $a \to b$ if a is a dependency of b and at least one honest device has sent or processed both a and b.

3. For each snapshot received by an honest device n, add a vertex r that represents the read of the snapshot by n. If n has sent any messages after the snapshot, add an edge $r \to t$ to the vertex t corresponding to the first such message.

4. For each snapshot received by an honest device n that was created by an honest device n', add an edge $m_s \to r$, where m_s is the message by n' defining the document state contained in the snapshot.

5. For each snapshot s received by an honest device n that was created by a faulty device, we add insert and delete operations by faulty devices as required to make the graph consistent as follows.

   Let src be the $deviceID$ of the device that created the snapshot $s = (A_s, sdesc, \_, \_)$.

   Let $(src, \_, \_, deps, \_, \_, \_, \_) \in sdesc$ be the state descriptor for src in s.

   For every honest device n' let $(n', ctr_{n'}, \_) \in deps$ be the snapshot's dependency on n', and $dep_{s,n'}$ be the vertex in $G_{n'}$ corresponding to the message from n' with sequence number $ctr_{n'}$.

   For each device n' we now find the set of operation messages that n' has observed by the time it produced $dep_{s,n'}$, and define the union of all these messages to be $ops(s)$:

$$ops(s) = \bigcup_{n'} \{m' \mid m' \preceq_{G_{n'}} dep_{s,n'}\} \tag{6.2}$$

   Let $S(msgs)$ be the set of atoms that have been inserted but not deleted within a set of operation messages $msgs$, as defined in (5.10).

   Let A_s be the set of atoms received as part of the snapshot.

   $M_n = \{(\_, src_a, \_, \_) \in A_s \setminus S(ops(s)) \mid src_a \text{ is faulty}\}$ is defined to be the set of atoms by faulty devices that are part of the snapshot but have not been seen by any honest device before the snapshot.

Now for each such snapshot s, do the following:

(a) For each message $dep_{s,n'}$ from an honest device referred to in *deps*, add an edge $dep_{s,n'} \to r$, where r is the snapshot read vertex added in point 3.

(b) For each atom $a \in M_n$, add a vertex i_a with an operation $\mathsf{insert}(a)$, an edge $dep_{s,n'} \to i_a$ for each honest device n', and an edge $i_a \to r$.

(c) For each atom $a \in S(ops(s)) \setminus A_s$, add a vertex d_a with an operation $\mathsf{delete}(a)$, an edge $dep_{s,n'} \to d_a$ for each honest device n', and an edge $d_a \to r$.

Lemma 6.4.1. G is a directed acyclic graph.

Proof. First observe that for any edge $a \to b$ where a is a dependency of b, a's version vector must be smaller than b's. For a snapshot created by an honest device, point 4 of the construction of G (Definition 6.7) adds a path between vertex m_s and the corresponding vertex r. For a snapshot created by a faulty device, point 5 adds a number of paths between vertices for dependencies $dep_{s,n'}$, and r. Since each r does not have outgoing edges except to the corresponding t (as defined in point 3), and $v(m_s) < v(t)$ and $v(dep_{s,n'}) < v(t)$, the invariant $v(a) < v(b)$ is preserved for all edges $a \to b$ between actual messages, and additional edges do not add cycles. □

Since two consecutive messages m_i, m_{i+1} sent by the same honest device always have a dependency relation between them, and $v(m_i) < v(m_{i+1})$, from the above proof it also follows that G is consistent with their real-time ordering. Thus, the FJC1 property holds for G.

Lemma 6.4.2. Let n be an honest device, let m be a vertex corresponding to a message in G_n, and let $m_{D,a}$ be a vertex containing a delete operation for an atom a. If $m_{D,a} \preceq_G m$, and n has processed m, then all insert operations for a processed by n precede m in G_n.

Proof. Let (src_a, ctr_a) be the source and counter of a, and let c_a be the entry for src_a in the version vector of m. Since the insertion of a must have happened before $m_{D,a}$ and therefore also before m, $ctr_a < c_a$. Due to the checks performed on sequence numbers, n does not accept an insert operation with a *ctr* less than or equal to the *ctr* of the latest message from src_a included in n's state. Thus, if n has already processed m, it will not accept an insert operation for a that happened either after or concurrent to m. □

Corollary 6.4.2.1. If $m_{D,a} \preceq_G m$, $m \in G_n$, and $a \notin read(G_n, m)$, then for any vertex m' that succeeds m in G_n ($m \prec_{G_n} m'$), $a \notin read(G_n, m')$.

Lemma 6.4.3. Let m be a vertex in G corresponding to a message, and let n be an honest device such that $m \in G_n$. Assume that for every honest device $\tilde{n}$ and every vertex r_n corresponding to a snapshot received by $\tilde{n}$, $read(G, r_{\tilde{n}}) = read(G_{\tilde{n}}, r_{\tilde{n}})$ (note that this

assumption will be proved in Lemma 6.4.4). Then for every such vertex m we have $read(G, m) = read(G_n, m)$.

Proof. By well-founded induction on m using the order relation $\prec_G$. That is, for any vertex m in G we assume the inductive hypothesis:

$$\forall m'.\ m' \prec_G m \implies read(G, m') = read(G_n, m') \tag{6.3}$$

and hence prove $read(G, m) = read(G_n, m)$. We break this down into two subgoals, $read(G_n, m) \subseteq read(G, m)$ and $read(G, m) \subseteq read(G_n, m)$.

$read(G_n, m) \subseteq read(G, m)$: Let a be an atom in $read(G_n, m)$. We start by showing that there must be an insert operation for a at or before m in G. Let $m_{I,a}$ be the message containing the insert operation for a observed by n. If $m_{I,a} = m$, $m_{I,a} \preceq_G m$ is trivially true. Otherwise, there exists at least one edge $d \to m$ in G_n such that $m_{I,a} \preceq_{G_n} d$, and there is no delete operation for a that precedes m in G_n. Thus, $a \in read(G_n, d)$. If d corresponds to an actual message, by the induction hypothesis, $a \in read(G_n, d) = read(G, d)$. Otherwise, d must be a vertex corresponding to a snapshot received by n, and we can apply the assumption $read(G, r_{\tilde{n}}) = read(G_{\tilde{n}}, r_{\tilde{n}})$ to conclude $a \in read(G_n, d) = read(G, d)$. Therefore, there must be an insert operation for a in G at or preceding d, $m'_{I,a} \preceq_G d \prec_G m$.

To show that $a \in read(G, m)$, it remains to be shown that there is no delete operation for a at or before m in G. Suppose there was a vertex $m_{D,a} \in G$ containing such a delete operation for a, with $m_{D,a} \preceq_G m$. We show that this contradicts $a \in read(G_n, m)$. If $m_{D,a} = m$, this directly contradicts $a \in read(G_n, m)$. Otherwise, at least one vertex d with a edge $d \to m$ must contain or succeed the delete operation in G, $m_{D,a} \preceq_G d$. Let $d_1, d_2, ..., d_k$ be all such vertices. We consider two cases, whether any such d_i is in G_n, or not.

Case $d_i \in G_n$ for some i. Let d be any such d_i. Since $m_{D,a} \preceq_G d$, $a \notin read(G, d)$. We now show that $d \to m$ also exists in G_n and that $a \notin read(G_n, d)$. If d is an actual message, d must be a dependency of m, and by definition of G_n, $d \to m$ must be present in G_n. By the induction hypothesis, $a \notin read(G, d) = read(G_n, d)$. Otherwise, d must be a vertex corresponding to a snapshot received by n. Since in this case, d does not exist in any other device's view, the edge $d \to m$ can only exist in G if it exists in G_n. Thus, $d \prec_{G_n} m$. By the assumption $read(G, r_{\tilde{n}}) = read(G_{\tilde{n}}, r_{\tilde{n}})$ we have $a \notin read(G, d) = read(G_n, d)$.

Since $m_{D,a} \preceq_G d$, and we can apply Corollary 6.4.2.1, which implies that $a \notin read(G_n, m)$.

Case $d_i \notin G_n$ for all i. Let d be any such d_i. We consider two cases: whether d corresponds to an actual message, or not.

Consider first the case where d corresponds to an actual message received by an honest device n'. Since $d \to m \in G$, d must correspond to one of m's dependencies. Thus, n must have processed d before processing m, unless m is the first message by n after a snapshot. Since $d \notin G_n$, the latter must be true. Since $a \notin \mathit{read}(G, d)$, by the induction hypothesis, $a \notin \mathit{read}(G_{n'}, d)$. Since $m_{D,a} \preceq_G d$, by Corollary 6.4.2.1, in n''s view, the document does not contain a at message m, i.e. $a \notin \mathit{read}(G_{n'}, m)$. Since m was created by an honest device, and both n and n' have processed it and compared the accumulator value to its view of the set of atoms, they agree on the set of atoms at m. Thus, $a \notin \mathit{read}(G_{n'}, m) = \mathit{read}(G_n, m)$.

Now consider the other case, where d is one of the vertices we added when constructing G. Since m is an actual message, and there exists a edge $d \to m$ in G, and $d \notin G_n$, d must be a vertex added for a snapshot processed by a different device n', and m must be the first message by n' after that snapshot. Since $m_{D,a} \preceq_G d$, by the construction of G, $m_{D,a}$ must either be included in one of the state descriptors for an honest device $\bar{n}$ contained in the snapshot (i.e. $m_{D,a} \preceq_G s_{\bar{n}}$, where $s_{\bar{n}}$ is the message corresponding to $\bar{n}$'s state descriptor), or $m_{D,a}$ must be an additional vertex added to G in our construction (Definition 6.7, point 5(c)). In both cases, a is not part of the snapshot received by n', and the insert operation for a must have happened before the snapshot. Let $(\mathit{src}_a, \mathit{ctr}_a)$ be the source and counter of a, and let c_a be the entry for src_a in the version vector associated with the snapshot. Again in both cases, $\mathit{ctr}_a \leq c_a$, and therefore, n' would not accept an insert operation for a since its ctr would conflict with the snapshot it has received. Therefore, in n''s view, the document does not contain a at message m, i.e. $a \notin \mathit{read}(G_{n'}, m)$. Since m was created by an honest device, both n and n' have processed m, and both devices behave correctly, m's accumulator value must match the set of atoms at both devices, i.e. $\mathit{read}(G_{n'}, m) = \mathit{read}(G_n, m)$. Thus, $a \notin \mathit{read}(G_n, m)$.

$\mathit{read}(G, m) \subseteq \mathit{read}(G_n, m)$: Let $a \in \mathit{read}(G, m)$. Thus, there exists a vertex $m_{I,a}$ with an insert operation for a such that $m_{I,a} \preceq_G m$, and no delete operation before or within m. If $m_{I,a} = m$, since a delete operation cannot have happened before the insertion, $a \in \mathit{read}(G_n, m)$. Otherwise, there exists at least one edge $d \to m$ such that $m_{I,a} \preceq_G d$, and thus $a \in \mathit{read}(G, d)$.

We consider two cases, whether $d \in G_n$, or not.

Case $d \in G_n$. Depending on whether d corresponds to an actual message or to a snapshot, we can apply the induction hypothesis or the precondition for snapshot vertices

$(read(G, r_{\tilde{n}}) = read(G_{\tilde{n}}, r_{\tilde{n}}))$, to conclude that $a \in read(G_n, d)$. Hence, there exists an insert operation for a in G_n preceding m.

Next we show that there is no delete operation for a before m in G_n. Assume, for the sake of contradiction that there was a message $m_{D,a} \prec_{G_n} m$ containing a delete operation for a. We show that this contradicts $a \in read(G, m)$.

We first consider the case that there exists a set of vertices $m_1, m_2, ..., m_k$ corresponding to actual messages such that $m_{D,a} \to m_1 \to ... \to m_k \to m$. By construction of G, the same set of messages and edges has to exist in G too, contradicting $a \in read(G, m)$.

Otherwise, if no such set of vertices exist, $m_{D,a}$ must be a vertex of the type added for a snapshot in point 6 of the construction of G_n (Definition 6.5). Let r_i be the vertex corresponding to the snapshot directly after $m_{D,a}$. The existence of the delete vertex implies that a is not part of the set of atoms in the snapshot corresponding to r_i, but there is a vertex $\tilde{m}$ corresponding to n's entry in the dependencies of the snapshot in G_n, where a was still present: $a \in read(G_n, \tilde{m})$. Thus, there is an edge $\tilde{m} \prec_G r_i$ in G. Since m has happened after r_i, it must have also happened after the first message $m_{n,i}$ by n after r_i, and we get $\tilde{m} \prec_{G_n} r_i \prec_{G_n} m_{n,i} \preceq_{G_n} m$. Furthermore, since $m_{n,i} \preceq_{G_n} m$, there is a path of vertices corresponding to messages by n, $m_{dev,i} \to \hat{m}_1 \to ... \to \hat{m}_l$ in G_n such that $\hat{m}_1$ is a dependency of m. Since the vertices correspond to messages processed by an honest device, the same path has to exist in G, and we get $r_i \prec_G m_{n,i} \preceq_G m$. By the precondition, $a \notin read(G_n, r_i)$ implies that $a \notin read(G, r_i)$, and since $a \in read(G_n, \tilde{m}) \overset{\text{IH}}{=} read(G, \tilde{m})$ and $\tilde{m} \prec_G r_i \prec_G m$, there must be an insert operation for a preceding r_i in G. Because $\tilde{m} \prec_G r_i$ and $a \notin read(G, r_i)$, there must be a delete operation for a preceding r_i in G. Since $r_i \prec_G m$, $a \notin read(G, m)$, reaching the desired contradiction.

Therefore, there is no delete operation for a before or at m in G_n, and since the edge $d \to m$ is present in G_n by construction, a must still be present at m in n's view. Thus, $a \in read(G_n, m)$.

Case $d \notin G_n$. As before, there are two cases. Either m is the first message by n after a snapshot (since n must have received all direct dependencies of any other message before processing it), or m is the first message by another honest device $\hat{n}$, and d is the vertex corresponding to the snapshot received by $\hat{n}$.

In the first case, since $d \in G$ and $d \to m \in G$, there must be at least one honest device n' that has processed d and m. By the induction hypothesis, $read(G_n, d) = read(G_{n'}, d)$, and therefore $a \in read(G_{n'}, d)$. For the same reasons as above, there cannot be a delete operation for a before or within m in G_n, and therefore a must still be present at m in n''s view. Since both n and n' are honest devices, and n'

must have verified the accumulator of m, they must have seen the same set of atoms at m, $a \in \mathit{read}(G_{n'}, m) = \mathit{read}(G_n, m)$.

In the latter case, by precondition, $a \in \mathit{read}(G_{\hat{n}}, d)$, implying that $m_{I,a} \preceq_{G_{\hat{n}}} d$. Again, there cannot be a delete operation for a before or at m in G, and thus a must still be present in $\hat{n}$'s view at m. Therefore, $a \in \mathit{read}(G_{\hat{n}}, m)$, and since n has agreed on the accumulator value of m and thus on the set of atoms, $a \in \mathit{read}(G, m)$.

□

Lemma 6.4.4. Let r_n be a vertex associated with a snapshot s received by n. Then the read corresponding to this vertex (which returns the atoms that were part of the snapshot) fulfils property FJC2 with regard to G, i.e. $\mathit{read}(G_n, r_n) = \mathit{read}(G, r_n)$.

Proof. By well-founded induction on r_n (using the happened-before ordering induced by G).

If n has created the document, the statement is trivially true. Otherwise, let $A_n = \mathit{read}(G_n, r_n)$, and $A = \mathit{read}(G, r_n)$. Let $\hat{n}$ be the device creating the snapshot. If $\hat{n}$ behaves correctly, this implies that the snapshot corresponds to a message m_s created by $\hat{n}$. By induction and Lemma 6.4.3, $\mathit{read}(G_{\hat{n}}, m_s) = \mathit{read}(G, m_s)$, and since the only incoming edge for r_n in both G and G_n is from m_s, $\mathit{read}(G_n, r_n) = \mathit{read}(G_{\hat{n}}, m_s) = \mathit{read}(G, m_s) = \mathit{read}(G, r_n)$.

If $\hat{n}$ is faulty, we first show that $A_n \subseteq A$. Let $a = (\mathit{src}, \mathit{ctr}_a, \_, \_) \in A_n$. Thus, a was part of the set of atoms presented as part of the snapshot. We define s_h to be the message by honest device h corresponding to the h's state descriptor. We consider three cases:

src **is an honest device.** Let $(\mathit{src}, \mathit{ctr}_{\mathit{src}}, \_, \_, \mathit{acc}, \mathit{mh}, \mathit{sig}, \mathit{wit})$ be the state descriptor for *src* presented in the snapshot, let m_{src} be the corresponding message from *src* with sequence number $\mathit{ctr}_{\mathit{src}}$, and let $m_{I,a}$ be the message containing the insert operation for a. The snapshot is only accepted by n if $\mathit{ctr}_a < \mathit{ctr}_{\mathit{src}}$. Using *sig*, *acc*, and *wit*, n has verified that $a \in \mathit{read}(G_{\mathit{src}}, m_{\mathit{src}}) \overset{\text{IH, Lemma 6.4.3}}{=} \mathit{read}(G, m_{\mathit{src}})$, and $m_{I,a} \prec_G m_{\mathit{src}} \prec_G m$. It remains to be shown that there is no delete operation for a in G before m.

For each honest device h that was part of the snapshot where s_h happened after $m_{I,a}$, n has verified using the Merkle consistency proofs that $m_{I,a} \prec_{G_h} s_h$, and it has verified using h's witness that $a \in \mathit{read}(G_h, s_h) \overset{\text{IH, Lemma 6.4.3}}{=} \mathit{read}(G, s_h)$, and thus there exists no delete operation for a before any s_h. Lastly, since a is part of the snapshot, no delete operation for a is added in point 5 of the construction of G (Definition 6.7).

src **is faulty.** We further consider two sub-cases: whether at least on one honest device h has observed an insert operation for a before s_{src}.

If yes, based on the dependencies in h's state descriptor, n can infer that the insertion has happened before s_h. Again this means that n has verified using h's witness that $a \in \mathit{read}(G_h, m_h)$. The rest of the argument is as in the previous case.

If not, since a is part of the snapshot, the construction of G, in particular point 5, ensures that there exists an insert operation for a before r_n, and no delete operation.

Now we show that $A \subseteq A_n$. Let $a \in A$. For the sake of contradiction, assume $a \notin A_n$. We consider two cases:

$a \in S(ops(s))$ **(as defined in G's construction).** This implies that in G, an operation $\mathsf{delete}(a)$ was added before r_n, contradicting $a \in A$.

$a \notin S(ops(s))$**.** This implies that either no honest device has seen an insert operation for a before r_n, or at least one has seen a delete operation for a. Either way, $a \notin A$.

□

Corollary 6.4.4.1. For a device n and a message $m \in G_n$, n's view of the document at m is equal to the state according to G, $read(G, m) = read(G_n, m)$. Thus, the privacy-enhanced protocol preserves FJC2.

6.5 Convergence and availability

For the basic protocol, convergence and availability directly follow from the properties of the CRDT and from the use of cryptographic hashes for dependencies. However, a fork-resolution protocol is required to resolve forks caused by misbehaving devices. Such a protocol is out of scope of this dissertation.

For the privacy-enhanced protocol, fork-join-causal consistency ensures that the views of honest group members converge to a consistent state. This again requires a fork-resolution protocol in case a misbehaving devices causes the views of honest devices to be forked. Any two participants can generally communicate even if other collaborators are offline; however, if multiple devices join concurrently, they require the help of an existing collaborator to reach a state where they can collaborate directly, since neither of them has seen all required dependencies of the others at the time of joining.

6.6 Discussion

The privacy-preserving variant of our protocol has a significant computational and metadata overhead. The costs seem reasonable for text editing with line-granularity atoms, especially since most of the expensive operations can be parallelized and typically can be run in the background without interrupting the editing process. However, for character-level granularity or similar, the costs seem prohibitive, in particular if the document is large and collaborators get added frequently, or when edits are performed at a high frequency. The protocol may be well suited for other types of collaborative applications such as shared calendars or to-do lists [KB17a].

While the protocol has a relatively large overhead, it scales well with the size of the document. Assuming a bounded number of devices and not considering costs for the CRDT metadata, communication and computation costs for editing operations are constant or amortized constant, except when a new collaborator is added, in which case the cost is (practically) linear in the number of atoms.

However, communication costs also grow with the number of operations due to the CRDT metadata. We used the Treedoc CRDT without optimizations, which generates a relatively large communication overhead for CRDT metadata because the tree is not balanced and every tree node stores a device identifier. To counter this, one can use a CRDT more optimized for the application, e.g. LSEQ [NMMD13] for text editing, and introduce device identifier compression. Furthermore, metadata overhead can be reduced by allowing a list of operations to be sent within a message instead of only a single operation per message.

To reduce the cost for insert operations and snapshot verification, if the prime representative is generated as described in Section 2.3.1, the device inserting the atom can include the last 2 bytes of the prime representative with the atom metadata. Other devices only need to verify its validity, but do not need to recompute it. Note that in this case it is not necessary that the smallest d is chosen, as long as the result is a prime and every device uses the same d.

Some information about the history can still be inferred from the metadata found in the privacy-enhanced scheme, in particular how many operations have been performed on each device, and the position identifiers and counters may allow some inferences about the positions where text fragments were deleted and how much was deleted. On the other hand, it may be desirable to know at which positions parts of the document have been deleted, as the device creating a snapshot can omit arbitrary atoms and therefore potentially completely change the meaning of the content. Metadata does not, in general, allow someone in possession of only a snapshot to infer positions where atoms have been deleted.

Lastly, our protocol relies on CRDTs where atoms have totally ordered position identifiers. More research is needed to add support for other operation-based CRDTs that do not have this property, like RGA [RJKL11].

6.7 Summary

In this chapter, we evaluated the performance of the protocol for authenticated snapshots presented in Chapter 5. We further revisited the correctness and security goals in Section 5.2.1, and showed how they are achieved. The performance evaluation was done based on editing histories of 270 Wikipedia pages, and showed that while it has a

significant computational overhead – primarily due to the use of RSA accumulators – its performance is reasonable if applied to small documents or using a coarse granularity (e.g. line-based rather than character-based). 99% of insert operations were processed within 11.0 ms, and 99% of delete operations within 64.9 ms. We also measured a median 84% reduction in the data transferred to a new collaborator by using authenticated snapshots compared to a basic protocol that transfers the full editing history. Therefore it may be well suited for applications such as shared calendars and to-do lists, where users tend to make relatively few edits, and a coarser granularity of edits may be acceptable. Further research is needed to make the protocol more practical for general real-time editing with character-level granularity.

CHAPTER 7

Conclusion

The aim of this work was to provide fundamentals for collaborative applications that are more decentralised in their architecture, with security, integrity, and privacy properties that rely less on a central server or authority. The focus was on two aspects: a decentralised and privacy-preserving communication channel between devices; and a protocol for authenticated snapshots for collaborative applications that hide the edit history from new collaborators without sacrificing integrity and consistency.

We considered using Tor and its hidden services as a decentralised and privacy-preserving communication medium between devices. In the Western world, smartphones are now the most widely used computing devices suitable for the kind of collaborative applications we considered, but also the most resource-constrained ones. We therefore evaluated the costs of running a Tor hidden service on a smartphone. We estimated that maintaining a hidden service would require a median of 198 MiB monthly cellular data traffic, assuming a typical usage pattern. The majority of this cost was due to regular downloads of the Tor network status consensus document. We then evaluated a number of optimisations to reduce cellular data usage. The strategy that was most effective based on our models – transferring diffs of the network status document instead of the full document each time – has since been integrated into the Tor code. Based on our estimates, this should roughly half the overall cellular data traffic to around a hundred MiB per month. With regards to battery usage, we estimated the network activity would cost at least 9.6% of battery capacity on a Nexus One connected to the Internet via 3G with a daily charge cycle.

With growing data allowances, the amount of cellular data traffic is becoming less of a concern, but it currently remains significant and will likely be an obstacle for wide adoption on smartphones in the near future. A wide adoption would be particularly desirable, since it would increase the size of the anonymity set of smartphone users. Another potential barrier to adoption is battery usage. More research is needed into battery usage on

modern smartphones. However, anecdotal evidence from developers and users of Briar – a smartphone application that already uses hidden services as a communication channel – shows that battery usage tends to be an important issue for users [akw15]. Additionally, during our experiments we found that the Tor code did not interact well with aggressive sleep policies in more recent Android versions, such as Doze. Addressing these problems and other open bugs will likely decrease battery usage. Reductions in network traffic will similarly have positive effects on battery usage.

An architecture with devices communicating peer-to-peer also requires that the communicating peers are connected via some kind of network, and ready to send and receive simultaneously. Research is needed to evaluate how this affects the synchronization delays for users with different kinds of devices and connectivity patterns, and for different kinds of applications and group sizes. Smart scheduling of communication between devices may help to reduce battery consumption and network traffic. Another approach to tackle this problem is to provide offline storage. This could be done using a decentralised distributed data store such as Freenet [CSWH01]. Loopix [PHE+17] is another promising anonymity network, which provides offline storage for users, but requires them to choose and connect to a fixed provider that stores messages for them. Another advantage of Loopix over Tor is that it is resistant against a global passive adversary. This comes at a cost of additional bandwidth costs for cover traffic, and an increased message delivery delay in the order of seconds, but these may be acceptable for some applications.

We also presented a protocol for real-time collaborative editing that supports authenticated snapshots. Snapshots allow collaborators to invite new collaborators and present to them the current state of the shared document, without revealing the past edit history. Authenticated snapshots allow a new collaborator to verify the consistency of their state with the views of other honest devices, even if other devices are faulty or malicious. We evaluated the costs of the protocol using the editing history of 270 Wikipedia pages, and compared it to a basic protocol without snapshots. We showed that the protocol is scalable for a bounded number of users. However, computational costs include a significant constant factor due to the use of RSA accumulators. In our experiments, 99% of insert operations were processed within 11.0 ms, and 99% of delete operations within 64.9 ms on a single core on a 2013 i5 desktop CPU. The computational cost for verifying a snapshots was, in practice, approximately linear in the number of atoms and collaborating devices; it took roughly half a millisecond per atom per device on average. On the other hand, snapshots can reduce the amount of data transfer necessary to add a new collaborator. We measured a median 84% reduction of data transferred compared to a basic protocol that sends the complete editing history to a new collaborator. We further showed how the protocol achieves the desired integrity, privacy, and availability properties.

The substantial computational costs likely make the protocol too costly for general real-time editing of documents on a character-granularity in the near future, in particular for resource-constrained devices like smartphones. More research is needed to determine how to reduce these costs; one possible direction could be to explore adaptive or hierarchical granularity of atoms. Another research question is whether more efficient constructions exist that can replace the RSA accumulators. However, while the computational costs are significant, they are not prohibitive for applications where edits are less frequent and where the number of atoms remains relatively small. The protocol may be well suited for applications such as shared to-do lists, photo albums, contact lists, or calendars.

Apart from improving efficiency, there are other interesting research directions to explore for protocols that provide authenticated snapshots. The protocol presented leaks some information about the editing history through metadata. One question is whether it is possible to reduce or eliminate this leak. Another question is whether one can design a protocol that not only hides the edit history of a document, but also the creator of individual atoms. Lastly, the protocol described can detect inconsistencies or forks between honest devices. However, we did not explain how to resolve such a situation once it is detected. Future work may consider suitable resolution mechanisms.

Apart from the challenges tackled in this dissertation, there are other areas that require more research before completely decentralised solutions can be deployed in practice. One such challenge is the trust establishment problem. Current methods for exchanging public keys between users either rely on centralised infrastructure, or require manual verification by users. Colleagues are currently investigating the use of gossiping on local networks as a decentralised approach for trust establishment.

A decentralised architecture also changes the way data is synchronized between devices. Since devices can be offline or partitioned, different devices can be in many different states before edits are merged. There is no longer just a linear history that is ordered by a central server; the history becomes a directed acyclic graph. In some kinds of applications, this may lead to semantic conflicts between concurrent changes, where manual resolution may be desirable. More research is needed on how to design user interfaces that help users deal with the additional complexity.

A limitation of our system design is that it does not provide an atomic operation to update an atom; an update is currently realised by performing a deletion followed by an insertion. This can also lead to conflicts, e.g. if two users concurrently, while being not being connected, change a word in a text document or the date of a calendar entry. One possible direction to address the problem of dealing with conflicting current edits is to investigate more elaborate, application-specific data representations. For example, source code could be stored as an abstract syntax tree, reducing the likelihood of conflicts, e.g. when code is refactored.

Bibliography

[ABG+16] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. Specification and Complexity of Collaborative Text Editing. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC 2016, pages 259–268. ACM, 2016.

[AHF14] Tanya Agrawal, David Henry, and Jim Finkle. JPMorgan hack exposed data of 83 million, among biggest breaches in history. `https://www.reuters.com/article/us-jpmorgan-cybersecurity/jpmorgan-hack-exposed-data-of-83-million-among-biggest-breaches-in-history-idUSKCN0HR23T20141003`, 2014. Accessed on 10 March 2019.

[akw15] akwizgran. Briar Issue #44 – Reduce battery consumption. `https://code.briarproject.org/briar/briar/issues/44`, 2015. Accessed on 10 March 2019.

[And] Optimize for Doze and App Standby. `https://developer.android.com/training/monitoring-device-state/doze-standby.html`. Accessed on 10 March 2019.

[BB04] Dan Boneh and Xavier Boyen. Short Signatures Without Random Oracles. In *Advances in Cryptology – EUROCRYPT 2004*, pages 56–73. Springer, 2004.

[BD93] Josh Benaloh and Michael De Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Advances in Cryptology – EUROCRYPT '93*, pages 274–285. Springer, 1993.

[BGG94] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental Cryptography: The Case of Hashing and Signing. In *Advances in Cryptology – CRYPTO '94*, pages 216–233. Springer, 1994.

[BP97] Niko Barić and Birgit Pfitzmann. Collision-Free Accumulators and Fail-Stop Signature Schemes Without Trees. In *Advances in Cryptology – EUROCRYPT '97*, pages 480–494. Springer, 1997.

[Bre00] Eric A. Brewer. Towards Robust Distributed Systems. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC 2000, page 7. ACM, 2000.

[CF13] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *Public-Key Cryptography – PKC 2013*, pages 55–72. Springer, 2013.

[Cha81] David L. Chaum. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[CHKO12] Philippe Camacho, Alejandro Hevia, Marcos Kiwi, and Roberto Opazo. Strong accumulators from collision-resistant hashing. *International Journal of Information Security*, 11(5):349–363, 2012.

[CKS09] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. An Accumulator Based on Bilinear Maps and Efficient Revocation for Anonymous Credentials. In *Public Key Cryptography – PKC 2009*, pages 481–500. Springer Berlin Heidelberg, 2009.

[CL02] Jan Camenisch and Anna Lysyanskaya. Dynamic Accumulators and Application to Efficient Revocation of Anonymous Credentials. In *Advances in Cryptology – CRYPTO 2002*, pages 61–76. Springer, 2002.

[CSWH01] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In Hannes Federrath, editor, *Designing Privacy Enhancing Technologies*, pages 46–66. Springer, 2001.

[CW09] Scott A. Crosby and Dan S. Wallach. Efficient Data Structures for Tamper-evident Logging. In *Proceedings of the 18th USENIX Security Symposium*, pages 317–334. USENIX Association, 2009.

[DAKS13] Stephen Doswell, Nauman Aslam, David Kendall, and Graham Sexton. Please Slow Down!: The Impact on Tor Performance from Mobility. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM 2013, pages 87–92. ACM, 2013.

[DDM03] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a Type III Anonymous Remailer Protocol. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP 2003, pages 2–15. IEEE, 2003.

[DGHS16] Benjamin Dowling, Felix Günther, Udyani Herath, and Douglas Stebila. Secure Logging Schemes and Certificate Transparency. In *Computer Security – ESORICS 2016*, pages 140–158. Springer, 2016.

[DHS15] David Derler, Christian Hanser, and Daniel Slamanig. Revisiting cryptographic accumulators, additional properties and relations to other primitives. In *Topics in Cryptology – CT-RSA 2015*, pages 127–144. Springer, 2015.

[DKAS15] Stephen Doswell, David Kendall, Nauman Aslam, and Graham Sexton. A longitudinal approach to measuring the impact of mobility on low-latency anonymity networks. In *Proceedings of the 2015 International Wireless Communications and Mobile Computing Conference*, IWCMC 2015, pages 108–113. IEEE, 2015.

[dMLP+12] Hermann de Meer, Manuel Liedel, Henrich C. Pöhls, Joachim Posegga, and Kai Samelin. Indistinguishability of one-way accumulators. Technical Report MIP-1210, Department of Informatics and Mathematics, University of Passau, 2012.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. Technical Report ADA465464, Naval Research Laboratory, Washington DC, 2004.

[DPSS16] David Derler, Henrich C. Pöhls, Kai Samelin, and Daniel Slamanig. A General Framework for Redactable Signatures and New Constructions. In *Information Security and Cryptology – ICISC 2015*, pages 3–19. Springer, 2016.

[DR10] John Day-Richter. What's different about the new Google Docs: Making collaboration fast. `https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html`, 2010. Accessed on 10 March 2019.

[EG89] Clarence A. Ellis and Simon J. Gibbs. Concurrency Control in Groupware Systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, volume 18, pages 399–407. ACM, 1989.

[FMMS17] Dennis Felsch, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. SECRET: On the Feasibility of a Secure, Efficient, and Collaborative Real-Time Web Editor. In *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security*, AsiaCCS 2017, pages 835–848. ACM, 2017.

[FZFF10] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group Collaboration Using Untrusted Cloud Resources. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI 2010, pages 337–350. USENIX Association, 2010.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *ACM SIGACT News*, 33(2):51–59, 2002.

[GRS99] David Goldschlag, Michael Reed, and Paul Syverson. Onion routing. *Communications of the ACM*, 42(2):39–41, 1999.

[GT96] Ceki Gulcu and Gene Tsudik. Mixing E-mail with Babel. In *Proceedings of the 1996 Symposium on Network and Distributed System Security*, SNDSS 1996, pages 2–16. IEEE, 1996.

[HE11] Yan Huang and David Evans. Private Editing Using Untrusted Cloud Services. In *Proceedings of the 31st IEEE International Conference on Distributed Computing Systems Workshops*, pages 263–272. IEEE, 2011.

[IMOR03] Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. Proving Correctness of Transformation Functions in Real-Time Groupware. In *ECSCW 2003*, pages 277–293. Springer, 2003.

[Int18] Internet access – households and individuals, Great Britain: 2018. `https://www.ons.gov.uk/peoplepopulationandcommunity/householdcharacteristics/homeinternetandsocialmediausage/bulletins/internetaccesshouseholdsandindividuals/2018`, 2018. Accessed on 10 March 2019.

[IROM06] Abdessamad Imine, Michaël Rusinowitch, Gérald Oster, and Pascal Molli. Formal design and verification of operational transformation algorithms for copies convergence. *Theoretical Computer Science*, 351(2):167–183, 2006.

[JMSW02] Robert Johnson, David Molnar, Dawn Song, and David Wagner. Homomorphic signature schemes. In *Topics in Cryptology – CT-RSA 2002*, pages 244–262. Springer, 2002.

[KB17a] Martin Kleppmann and Alastair R. Beresford. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, 2017.

[KB17b] Stephan A. Kollmann and Alastair R. Beresford. Supporting data for "The Cost of Push Notifications for Smartphones using Tor Hidden Services". https://doi.org/10.17863/CAM.7547, 2017.

[KB17c] Stephan A. Kollmann and Alastair R. Beresford. The Cost of Push Notifications for Smartphones Using Tor Hidden Services. In *Proceedings of the 2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 76–85. IEEE, 2017.

[Kho17] Dara Khosrowshahi. 2016 Data Security Incident. https://www.uber.com/newsroom/2016-data-incident, 2017. Accessed on 10 March 2019.

[Kob18] Nadim Kobeissi. Capsule: A Protocol for Secure Collaborative Document Editing. IACR Cryptology ePrint 2018/253, 2018.

[Kol19] Stephan A. Kollmann. Code supporting "Snapdoc: Authenticated snapshots with history privacy in peer-to-peer collaborative editing". https://doi.org/10.17863/CAM.39062, 2019.

[LLK13] Ben Laurie, Adam Langley, and Emilia Kasper. RFC 6962: Certificate Transparency. IETF, 2013.

[LLS04] Rui Li, Du Li, and Chengzheng Sun. A time interval based consistency control algorithm for interactive groupware applications. In *Proceedings of the Tenth International Conference on Parallel and Distributed Systems*, ICPADS 2004, pages 429–436. IEEE, 2004.

[LLW09] Jörg Lenhard, Karsten Loesing, and Guido Wirtz. Performance Measurements of Tor Hidden Services in Low-Bandwidth Access Networks. In *Applied Cryptography and Network Security*, ACNS 2009, pages 324–341. Springer, 2009.

[LM07] Jinyuan Li and David Maziéres. Beyond One-third Faulty Replicas in Byzantine Fault Tolerant Systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, NSDI 2007, pages 131–144. USENIX Association, 2007.

[LSWW08] Karsten Loesing, Werner Sandmann, Christian Wilms, and Guido Wirtz. Performance Measurements and Statistics of Tor Hidden Services. In *Proceedings of the 2008 International Symposium on Applications and the Internet*, SAINT 2008, pages 1–7. IEEE, 2008.

[MAD11] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, Availability, and Convergence. Technical Report UTCS TR-11-22, Department of Computer Science, The University of Texas at Austin, 2011.

[Mar14] Daniel Martí. [GSoC] Consensus diffs - Fourth report. `https://lists.torproject.org/pipermail/tor-dev/2014-July/007163.html`, 2014. Accessed on 10 March 2019.

[MB09] Prateek Mittal and Nikita Borisov. ShadowWalker: Peer-to-peer Anonymous Communication Using Redundant Structured Topologies. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS 2009, pages 161–172. ACM, 2009.

[MBB+15] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: Bringing Key Transparency to End Users. In *Proceedings of the 24th USENIX Security Symposium*. USENIX Association, 2015.

[MCPS04] Ulf Möller, Lance Cottrell, Peter Palfrader, and Len Sassaman. Mixmaster Protocol Version 2. `https://tools.ietf.org/html/draft-sassaman-mixmaster-03`, 2004. Accessed on 10 March 2019.

[Mer88] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology – CRYPTO '87*, pages 369–378. Springer, 1988.

[MOT+11] Prateek Mittal, Femi Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. PIR-Tor: Scalable Anonymous Communication Using Private Information Retrieval. In *Proceedings of the 20th USENIX Security Symposium*. USENIX Association, 2011.

[MSL+11] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud Storage with Minimal Trust. *ACM Transactions on Computer Systems*, 29(4):12:1–12:38, 2011.

[MTHK09] Jon McLachlan, Andrew Tran, Nicholas Hopper, and Yongdae Kim. Scalable Onion Routing with Torsk. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS 2009, pages 590–599. ACM, 2009.

[MUW10] Stéphane Martin, Pascal Urso, and Stéphane Weiss. Scalable XML Collaborative Editing with Undo. In *On the Move to Meaningful Internet Systems: OTM 2010*, pages 507–514. Springer, 2010.

[NCDL95] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System. In *Proceedings of the 8th Annual ACM Symposium on User Interface Software and Technology*, UIST 1995, pages 111–120. ACM, 1995.

[Ngu05] Lan Nguyen. Accumulators from Bilinear Pairings and Applications. In *Topics in Cryptology – CT-RSA 2005*, pages 275–292. Springer, 2005.

[NMMD13] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing. In *Proceedings of the 2013 ACM Symposium on Document Engineering*, DocEng 2013, pages 37–46. ACM, ACM, 2013.

[NW06] Arjun Nambiar and Matthew Wright. Salsa: A Structured Approach to Large-Scale Anonymity. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS 2006. ACM, 2006.

[OUMI05] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Proving correctness of transformation functions in collaborative editing systems. Technical Report RR-5795, INRIA, 2005.

[Pal08] Peter Palfrader. Provide diffs between consensuses. `https://gitweb.torproject.org/torspec.git/tree/proposals/140-consensus-diffs.txt`, 2008. Accessed on 10 March 2019.

[Per17] Nicole Perlroth. All 3 Billion Yahoo Accounts Were Affected by 2013 Attack. `https://www.nytimes.com/2017/10/03/technology/yahoo-hack-3-billion-users.html`, 2017. Accessed on 10 March 2019.

[Per18] Jen Person. Time to Upgrade from GCM to FCM. `https://firebase.googleblog.com/2018/04/time-to-upgrade-from-gcm-to-fcm.html`, 2018. Accessed on 10 March 2019.

[PHE+17] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The Loopix Anonymity System. In *Proceedings of the 26th USENIX Security Symposium*. USENIX Association, 2017.

[PK01] Andreas Pfitzmann and Marit Köhntopp. Anonymity, Unobservability, and Pseudonymity – A Proposal for Terminology. In *Designing Privacy Enhancing Technologies*, pages 1–9. Springer, 2001.

[PMSL09] Nuno Preguiça, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A Commutative Replicated Data Type for Cooperative Editing. In *29th International Conference on Distributed Computing Systems*, ICDCS 2009, pages 395–403. IEEE, IEEE, 2009.

[PRR09] Andriy Panchenko, Arne Rache, and Stefan Richter. NISAN: Network Information Service for Anonymization Networks. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS 2009. ACM, 2009.

[PSP12] Henrich C. Pöhls, Kai Samelin, and Joachim Posegga. Length-Hiding Redactable Signatures from One-Way Accumulators in O(n). Technical Report MIP-1201, Department of Informatics and Mathematics, University of Passau, 2012.

[Rab80] Michael O. Rabin. Probabilistic Algorithm for Testing Primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[Rib04] Paulo Ribenboim. *The Little Book of Bigger Primes*. Springer, 2004.

[RJKL11] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, 2011.

[RP04] Marc Rennhard and Bernhard Plattner. Practical Anonymity for the Masses with MorphMix. In *Financial Cryptography*, FC 2004, pages 233–250. Springer, 2004.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[SB13] Petter Solberg and Bram Bezem. Performance of hidden services in Tor. Master's thesis, Norwegian University of Science and Technology, Department of Telematics, 2013.

[SBZ01] Ron Steinfeld, Laurence Bull, and Yuliang Zheng. Content Extraction Signatures. In *Information Security and Cryptology – ICISC 2001*, pages 285–304. Springer, 2001.

[Sin10] Nilotpal K. Sinha. On a new property of primes that leads to a generalization of Cramér's conjecture. *arXiv preprint arXiv:1010.1399*, 2010.

[SJZ+98] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems. *ACM Transactions on Computer-Human Interaction*, 5:63–108, 1998.

[SPBZ11a] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical Report RR-7506, INRIA, 2011.

[SPBZ11b] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, SSS 2011, pages 386–400. Springer, 2011.

[Tam03] Roberto Tamassia. Authenticated data structures. In *Algorithms – ESA 2003*, pages 2–5. Springer, 2003.

[Tora] Tor Metrics – Servers. `https://metrics.torproject.org/networksize.html`. Accessed on 10 March 2019.

[Torb] Tor Rendezvous Specification. `https://gitweb.torproject.org/torspec.git/tree/rend-spec-v2.txt`. Accessed on 10 March 2019.

[Torc] Tor Rendezvous Specification – Version 3. `https://gitweb.torproject.org/torspec.git/tree/rend-spec-v3.txt`. Accessed on 10 March 2019.

[tord] Tor Ticket #13339 – Prop140: Complete Consensus diffs / Merge GSoC project. `https://trac.torproject.org/projects/tor/ticket/13339`. Accessed on 10 March 2019.

[tore] Tor Ticket #16387 – Improve reachability of hidden services on mobile phones. `https://trac.torproject.org/projects/tor/ticket/19522`. Accessed on 10 March 2019.

[torf] Tor Ticket #19522 – HS intro circuit retry logic fails when network interface is down. `https://trac.torproject.org/projects/tor/ticket/19522`. Accessed on 10 March 2019.

[torg] Tor Ticket #8239 – Hidden services should try harder to reuse their old intro points. `https://trac.torproject.org/projects/tor/ticket/8239`. Accessed on 10 March 2019.

[VCFS00] Nicolas Vidot, Michelle Cart, Jean Ferrié, and Maher Suleiman. Copies Convergence in a Distributed Real-time Collaborative Environment. In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*, CSCW 2000, pages 171–180. ACM, 2000.

[VNTP14] Ekhiotz Jon Vergara, Simin Nadjm-Tehrani, and Mihails Prihodko. Energybox: Disclosing the wireless transmission energy cost for mobile devices. *Sustainable Computing: Informatics and Systems*, 4(2):118–135, 2014.

[WJ02] James Weatherall and Alan Jones. Ubiquitous networks and their applications. *IEEE Wireless Communications*, 9(1):18–29, 2002.

[WMLT15] David Wang, Alex Mah, Soren Lassen, and Sam Thorogood. Apache Wave (incubating) Protocol Documentation, Release 0.4. `https://people.apache.org/~al/wave_docs/ApacheWaveProtocol-0.4.pdf`, 2015. Accessed on 10 March 2019.

[WRB14a] Daniel T. Wagner, Andrew Rice, and Alastair R. Beresford. Device Analyzer: Large-scale mobile data collection. *ACM SIGMETRICS Performance Evaluation Review*, 41(4):53–56, 2014.

[WRB14b] Daniel T. Wagner, Andrew Rice, and Alastair R. Beresford. Device Analyzer: Understanding Smartphone Usage. In *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, pages 195–208. Springer, 2014.

[WSSN07] Rungrat Wiangsripanawan, Willy Susilo, and Rei Safavi-Naini. Achieving Mobility and Anonymity in IP-Based Networks. In *Cryptology and Network Security*, CANS 2007, pages 60–79. Springer, 2007.

[WUM09] Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In *29th IEEE International Conference on Distributed Computing Systems*, ICDCS 2009, pages 404–412. IEEE, IEEE, 2009.

[WUM10] Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot-Undo: Distributed Collaborative Editing System on P2P Networks. *IEEE Transactions on Parallel and Distributed Systems*, 21(8):1162–1174, 2010.

[XS16] Yi Xu and Chengzheng Sun. Conditions and Patterns for Achieving Convergence in OT-Based Co-Editors. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):695–709, 2016.

APPENDIX A

List of Wikipedia pages used for experiments in Chapter 6

1981_Grand_Marnier_Tennis_Games
1995_Stockholm_Open_–_Singles
2001_French_rugby_league_tour_of_New_Zealand_and_Papua_New_Guinea
2001_World_Archery_Championships_-_Men's_Individual_Recurve
2010_US_Open_–_Wheelchair_Women's_Doubles
2012–13_Appalachian_State_Mountaineers_men's_basketball_team
2017_Americas_Challenge
2018_Irving_Tennis_Classic_–_Singles
Ağamalılar
Acrosticta_rufiventris
Addicted_to_Love_(TV_series)
Adolf_Dieudonné
A_Double-Threaded_Life
Agnes_Zurowski
Alfie_Joey
A_Life_Force
All-Union_Communist_Party_of_Bolsheviks_(1991)
Al_Noor_Hospitals
Ammeldingen_bei_Neuerburg
Anglicisation_of_names
Antaramej
Argenton,_Lot-et-Garonne
Aristide_Caradja
Armando_Rossi

Association_for_Perioperative_Practice
Baggböle_Manor
Bagirmi_language
Bakhtiyar_Khilji's_Tibet_campaign
Baqershahr
Bardeh_Rashan,_Bukan
Basil_Coad
Behind_Locked_Doors
Ben_Small
Benwee_Head
Bernardia
Bernold
Big-eared_brown_bat
Bingham_Wapentake
Blood,_Guts,_Bullets_and_Octane
BlowOut
Bolivia–Denmark_relations
Boxing_at_the_2013_Mediterranean_Games_–_Men's_light_heavyweight
Brave_Rabbit_2_Crazy_Circus
Braxton_Family_Christmas
Broo_Brewery
Brooks_Public_Library
Buchanania_arborescens
Burgum
Burila_Mare
Árneshreppur
Éraville
Étienne_Lamotte
Cadwallon_ap_Gruffydd
Canton_of_Trélissac
Capital_Athletic_Conference_Men's_Basketball_Tournament
Cardamine_impatiens
Cards_on_the_Table_(Vietnamese_telefilm)
Casey_Radio
Central_District_(Dorud_County)
CFL_Class_3600
Chapel_Hill_(Mint_Spring,_Virginia)
Charles_Dewey_Day

Charlie_Shoemake
Cheri_Madsen
Chris_Wakelin
Clayton_County_Courthouse
Coalescence_(physics)
Collared_palm_thrush
Como,_Colorado
Confidence,_West_Virginia
Cross-country_skiing_at_the_1936_Winter_Olympics
David_Starr,_Space_Ranger
Düsseldorf-Wersten
Dera_Din_Panah
Divergence_(statistics)
Do_G's_Get_to_Go_to_Heaven?
Drilosphere
Dušan_Petričić
Duotone
East_and_West_Blockhouses
Ectoedemia_longicaudella
Edmonton_municipal_election,_1898
Elachista_semnani
Endre_Elekes
Ethmia_kabulica
F2RL2
Filip_Chlup
Firuzabad_District_(Kermanshah_Province)
Four_Great_Towers_of_China
Freddy_Martín
Fredrik_Johan_Wiik
Fred_Smye
Freedom_of_Information_and_Protection_of_Privacy_Act_(Nova_Scotia)
Fuster
Game_Critter_Super-Squad!
Ganda_Singhwala_railway_station
George_Marsden_(boxer)
Gilbert_Houngbo
Giovanni_Fouchetti
Good_King_Dagobert

Gore_Metal_(album)
Grand_Orient_of_Russia's_Peoples
Grant_Michaels
Greater_Lebanon
GT8
Gus:_The_Theatre_Cat
Gymnastics_at_the_1924_Summer_Olympics_–_Men's_rings
Halophanes
Haridas_Chaudhuri
Henriett_Koósz
Henry_Holland_(mayor)
Henry_Stokes_Tiffen
Herbert_Koch
Hilal_Al-Makdesi
Hiroko_Saito
Hizaj
HMS_Mackerel
Hoddevik
Holywell_Press
Hotel_El_Convento
House_of_Stolberg
Hypolycaena_similis
Hypothetical_astronomical_object
If-by-whiskey
Ilario_Pegorari
I_P_Mission_School
Ivan_Mikhailovich_Obolensky
Jacques-Émile_Blanche
James_Charles_Dalbiac
James_del_Piano
James_Joseph_Hines
Jangsan_station
Jason_Hammel_(musician)
Jedlnica
Jimmy_Lahoud
Johan_Nygaardsvolds_plass
John_Duncan_(neuroscientist)
John_Kay_(spinning_frame)

John_Peter_Barnes
John_Schappert_(video_game_executive)
Julia_Pierson_Emmet
Karan_Higdon
Kargilik_Town
Karnataka_State_Film_Award_for_Best_Child_Actor_(Female)
Kelp_goose
Khaled_Soliman
Kiki_Strike
Knox_Helm
Krasino,_Arkhangelsk_Oblast
Krayem_Awad
Krokosua_Hills_Forest_Reserve
Kumalo
Lakenheath_Fen_RSPB_reserve
Laramie_Boomerang
Laurence_Olivier_Award_for_Supporting_Artist_of_the_Year
Liezen_District
Linderpur
List_of_Billboard_number-one_country_albums_of_1980
List_of_English_Twenty20_cricket_champions
List_of_state_leaders_in_1357
List_of_Wyoming_companies
Livens_Projector
Loot_system
Lorraine_Loots
Lužce
Ludovico_Balbi
Luxiol
Małecz
Margarites_olivaceus
Matthew_5:22
Medeas
Medicago_murex
Megachile_minor
Michel_Clair
Mildred_Brown_Schrumpf
Mohammad_Yousuf_(cricketer,_born_1935)

Monte_Verde,_Cape_Verde
Monumento_a_la_Mujer
Munkustrap
NCK2
Never_Turn_Away
Newell,_California
Newgrange_(song)
Nieuport_Memorial
Northwest_Territories_general_election,_1954
Nova_Scotia_Route_205
Nowy_Lindów
Ochlochaete
Old_Calendarist_Romanian_Orthodox_Church
Otterwisch
Our_Lady_of_Lourdes_Medical_Center
Pancreatic_stellate_cell
Pay_the_Butler
Pedro_de_Araújo
Pegaso_Z-403
Pequea_Township,_Lancaster_County,_Pennsylvania
Perry,_Arkansas
Peter_I_of_Bulgaria
Photon_surface
Pig_pickin'
Politics_of_Transnistria
Posterior_labial_nerves
Professional_boxing_in_New_Zealand
Rajinder_Singh_(wrestler)
Ralph_Abercrombie_(public_servant)
Randegg
Ravna_(Knjaževac)
Reading_First
Reksio
Release_(agency)
Ricardo_Zamora
Rick_MacInnes-Rae
Right_to_withdraw
Robbie_Venter

Robert_de_Ferrers,_1st_Earl_of_Derby
Robert_Niven_(New_Zealand_cricketer)
Roman_Catholic_Diocese_of_Wheeling–Charleston
Rome_Kirby
Sacramento_Islamic_Mosque
Sarcolaena_oblongifolia
Sarola_Brahmin
Sarvodaya_Shramadana_Movement
Sérgio_Dias_Branco
Scenes_from_the_Passion_of_Christ
Seehorn,_Illinois
Sendets,_Pyrénées-Atlantiques
Shut_Up_and_Dance:_Mixes
Sid_Applebaum
Smartbomb_(book)
South_African_Class_24_2-8-4
Still_Falling_in_Love
Stuart_Hamblen
Suzuka_Hasegawa
Svelgen
Sym_River
Tarachodes_maurus
Tarong_North_Power_Station
Tentacled_dragonet
Terrington
The_Gift_(Douglas_novel)
The_Leake_County_Revelers
Themes_(Vangelis_album)
The_Rahway_Murder_of_1887
The_Roxy_(Rathbone_Place)
The_Wages_of_Sin_(Upstairs,_Downstairs)
ThinkServer
Thiruvidandai
Thomas_Blakiston
Thomas_Smedley_House
Thomas_Wright_(social_commentator)
Three_for_the_Money
Timeline_of_Orlando,_Florida

Tor_Miller
Toyin_Saraki
Trevor_the_Traction_Engine
Tropimetopa_simulator
Twisted_Hessian_curves
Ube_Industries
United_Services_Recreation_Club
Unley,_South_Australia
Van_Lieshout
Verckys_Kiamuangana_Mateta
Virginia_Allen_Crockford
Washington_Mustangs
Whose_Doctor_Who
X-fast_trie
Xia_Qin
Yehoshua_Feigenbaum
YOYOW
Zaghmar